50

TRƯỜNG ĐẠI HỌC CÔNG NGHỆ - ĐHQGHN

BÙI THẾ DUY

ĐỒ HỌA MÁY TÍNH

NHÀ XUẤT BẢN ĐẠI HỌC QUỐC GIA HÀ NỘI

ĐẠI HỌC QUỐC GIA HÀ NỘI
TRƯỜNG ĐẠI HỌC CÔNG NGHỆ

Bùi Thế Duy

ĐỒ HOẠ MÁY TÍNH

NHÀ XUẤT BẢN ĐẠI HỌC QUỐC GIA HÀ NỘI

MỤC LỤC

LỜI NÓI ĐẦU

Đồ họa máy tính là một lĩnh vực đang được quan tâm nhiều bởi các nhà nghiên cứu và các nhà phát triển phần mềm bởi sự đóng góp mạnh mẽ của nó trong lĩnh vực Công nghệ thông tin. Ngày nay, chúng ta có thể thấy sự xuất hiện của Đồ họa máy tính trong hầu hết các ứng dụng phần mềm. Các giao diện phần mềm dựa trên Đồ họa máy tính đã tạo nên cả một cuộc cách mạng về cách thức giao tiếp giữa con người và máy tính. Ngoài ra Đồ họa máy tính còn tạo nên những biến đổi sâu sắc trong nhiều lĩnh vực như điện ảnh, giáo dục và thiết kế kỹ thuật. Ngay từ khi mới phát triển vào những năm 1960-1970, Đồ họa máy tính đã được giảng dạy ở các trường đại học trên thế giới. Đây là một trong những môn học nền tảng cho ngành Công nghệ thông tin/Khoa học máy tính.

Cuốn sách "Đồ họa máy tính" được biên soạn dựa trên đề cương môn Đồ họa máy tính thuộc chương trình đào tạo ngành Công nghệ thông tin của Trường Đại học Công nghệ, Đại học Quốc gia Hà Nội. Ngoài những vấn đề cơ bản nhất của Đồ họa máy tính, nội dung của cuốn sách còn cung cấp thêm một số kiến thức nâng cao, cập nhật với Đồ họa máy tính hiện đại, đồng thời tập trung hướng đến yếu tố thực hành. Các thuật toán trong cuốn sách được minh họa bằng các đoạn chương trình C++, giúp cho người đọc có thể dễ dàng triển khai phần cài đặt của các thuật toán này. Cuốn sách cũng cung cấp kiến thức cơ sở để lập trình đồ họa máy tính ba chiều dựa trên bộ thư viện đồ họa đa nền tảng OpenGL. Cuốn sách có thể sử dụng làm tài liệu giảng dạy hoặc tham khảo cho môn "Đồ họa máy tính" dùng cho sinh viên đại học và cao học các ngành Công nghệ thông tin và kỹ thuật. Cuốn sách cũng rất hữu ích đối với những người nghiên cứu về Đồ họa máy tính.

Các nội dung trình bày trong cuốn sách được tham khảo từ nhiều tài liệu phổ biến trên thế giới về Đồ họa máy tính. Tuy nhiên, trong quá trình biên soạn không tránh khỏi những sơ suất, chúng tôi xin trân trọng tiếp thu tất cả những ý kiến đóng góp để cuốn sách ngày càng được hoàn thiện hơn. Cuối cùng, chúng tôi xin gửi lời cảm ơn chân thành đến những đồng nghiệp đã góp phần hỗ trợ cho sự ra đời của cuốn sách, đặc biệt là Đào Minh Thư, Ma Thị Châu và Nguyễn Duy Khương.

Tác giả

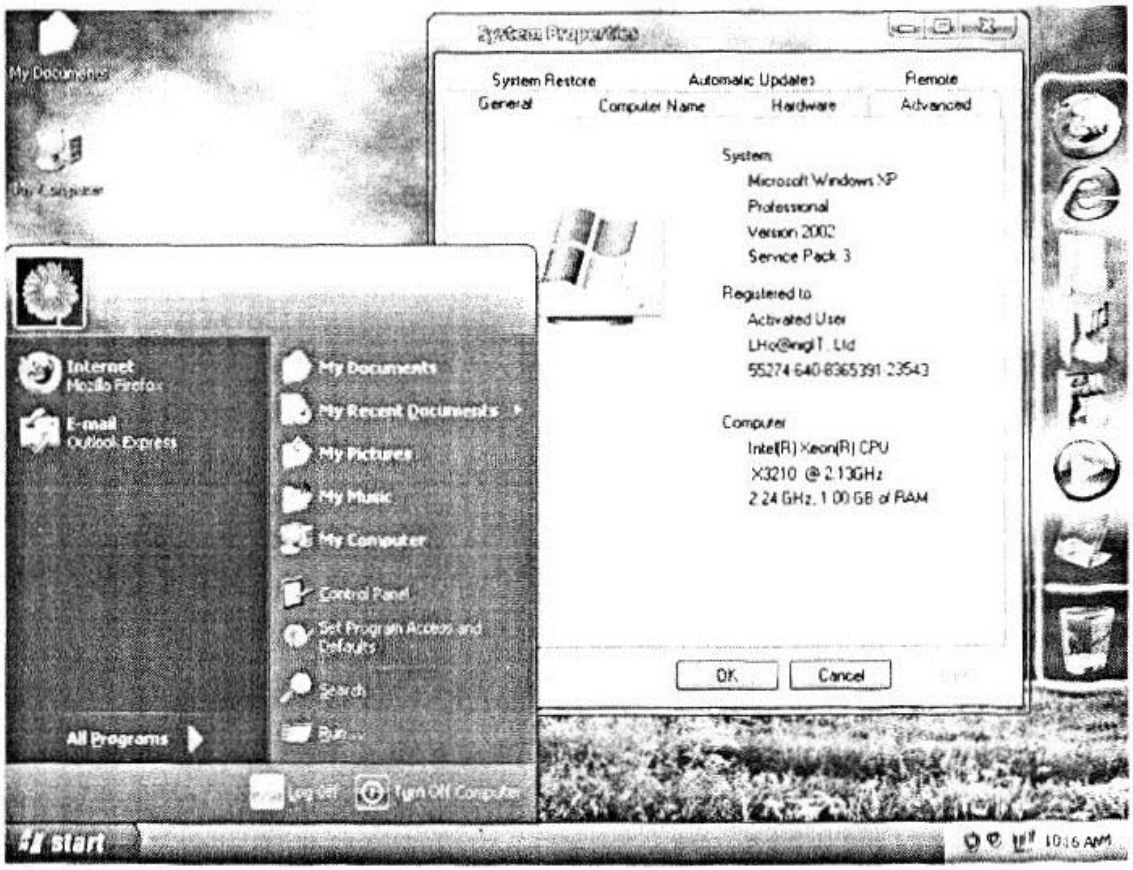
Chương 1
GIỚI THIỆU VỀ ĐỒ HỌA MÁY TÍNH

Khái niệm **Đồ họa máy tính** (*Computer Graphics*) được đưa ra vào năm 1960 bởi William Fetter để mô tả các kỹ thuật thiết kế mới sử dụng máy tính được thực hiện tại Boeing. Cụ thể hơn, đồ họa máy tính thường được hiểu là sự tạo ra, lưu trữ và thao tác với các bức ảnh nhân tạo dựa trên mô tả hoặc mô hình. Một lĩnh vực khá liên quan đến đồ họa máy tính là **Xử lý ảnh** (*Image Processing*). Tuy nhiên, khác với đồ họa máy tính, xử lý ảnh liên quan đến các kỹ thuật phân tích, tăng cường, nén và khôi phục ảnh, và thường là trên ảnh chụp được số hóa thông qua thiết bị quét.

Ngày nay, đồ họa máy tính đã được dùng trong rất nhiều lĩnh vực của công nghiệp, kinh doanh, hành chính, giáo dục và giải trí. Số lượng ứng dụng của đồ họa máy tính là rất lớn và đang ngày càng tăng lên khi các máy tính với khả năng xử lý đồ họa cao đang trở thành mặt hàng tiêu dùng bình thường. Sau đây là một số ứng dụng thông dụng.

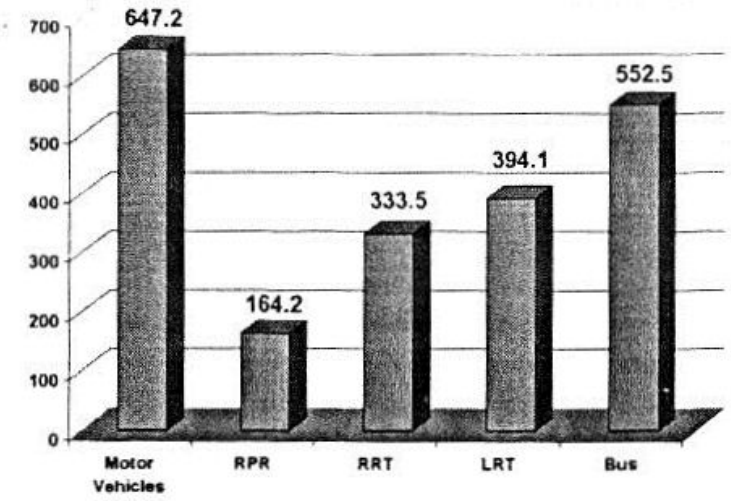
Giao diện người dùng. Có thể chúng ta không để ý đến việc chúng ta đang hàng ngày sử dụng các ứng dụng đồ họa máy tính. Đó chính là giao diện người dùng của hệ điều hành từ khi hệ điều hành với giao diện đồ họa ra đời ví dụ như Macintosh OS và Windows 3.1. Sau đó, hầu hết các ứng dụng trên máy tính cá nhân đã có các giao diện đồ họa dựa trên hệ thống cửa sổ với khả năng điều khiển bấm và trỏ bằng chuột. Chương trình soạn thảo văn bản và bảng tính là những ứng dụng kinh điển tận dụng được lợi thế của đồ họa máy tính. Hình 1.1 cho thấy giao diện của hệ điều hành

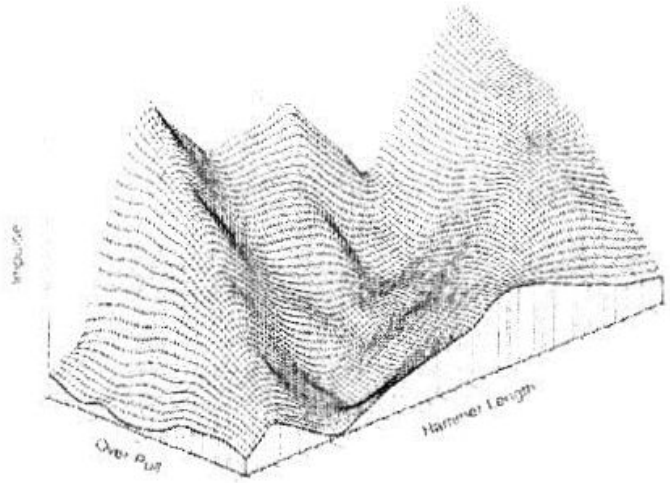
Microsoft Windows XP sử dụng những công nghệ tiên tiến trong đồ họa máy tính.



Hình 1.1. Giao diện của hệ điều hành Microsoft Windows XP sử dụng những công nghệ tiên tiến trong đồ họa máy tính.

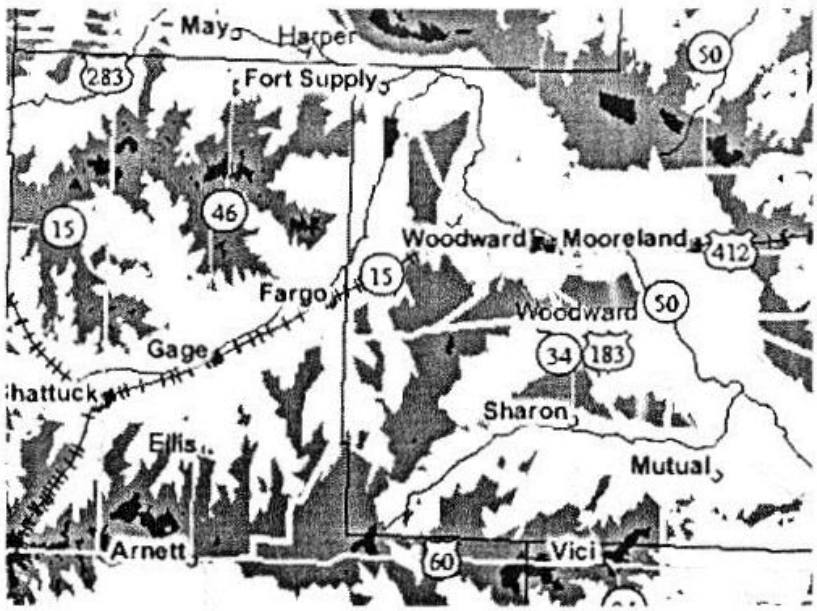
Vẽ biểu đồ trong kinh doanh, khoa học và công nghệ. Ứng dụng thông dụng thứ hai của đồ họa máy tính là việc tạo ra các đồ thị 2D và 3D của các hàm toán học, vật lý, và kinh tế; và các loại biểu đồ khác nhau (xem Hình 1.2).





Hình 1.2. Đồ họa máy tính được dùng nhiều để tạo ra các biểu đồ.

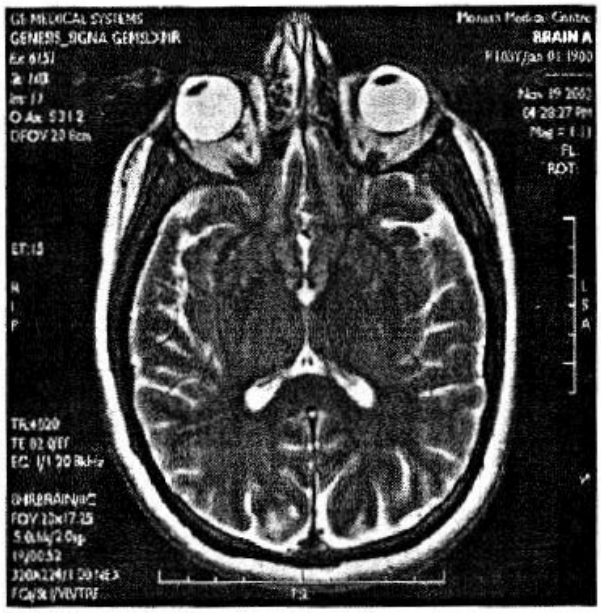
Bản đồ. Đồ họa máy tính đã được dùng để tạo ra những thể hiện chính xác và có hệ thống về địa lý cũng như những hiện tượng tự nhiên từ dữ liệu thu được. Ví dụ bao gồm các bản đồ địa lý, bản đồ tài nguyên và khoáng sản, các biểu đồ về đại dương, các bản đồ thời tiết, các bản đồ về dân số, ... (Hình 1.3).



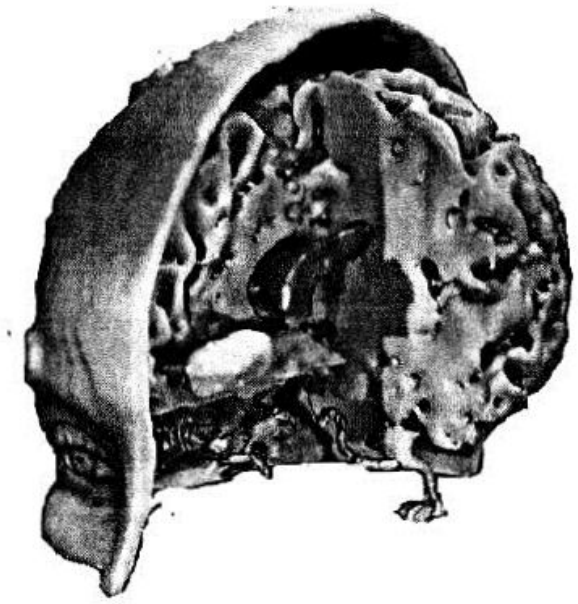
Hình 1.3. Các bản đồ địa lý, bản đồ khoáng sản, ...
được thể hiện chính xác với đồ họa máy tính.

Y học. Đồ họa máy tính đóng vai trò rất quan trọng trong chẩn đoán y học cũng như hỗ trợ phẫu thuật. Các bác sỹ phẫu thuật có thể dùng đồ họa để điều khiển các thiết bị một cách chính xác và cắt bỏ những mô cần loại bỏ. Một ví dụ về đồ họa máy tính trong

chẩn đoán y học được cho thấy trong Hình 1.4. Trong hình này, chúng ta thấy một ảnh của não người được dựng lên với dữ liệu từ máy chụp cộng hưởng từ (MRI). Từ rất nhiều lát chụp 2D, người ta có thể dựng lên thể hiện 3D của não (xem Hình 1.5). Người sử dụng có thể tương tác với mô hình 3D này để xem xét kỹ lưỡng tình trạng của não.

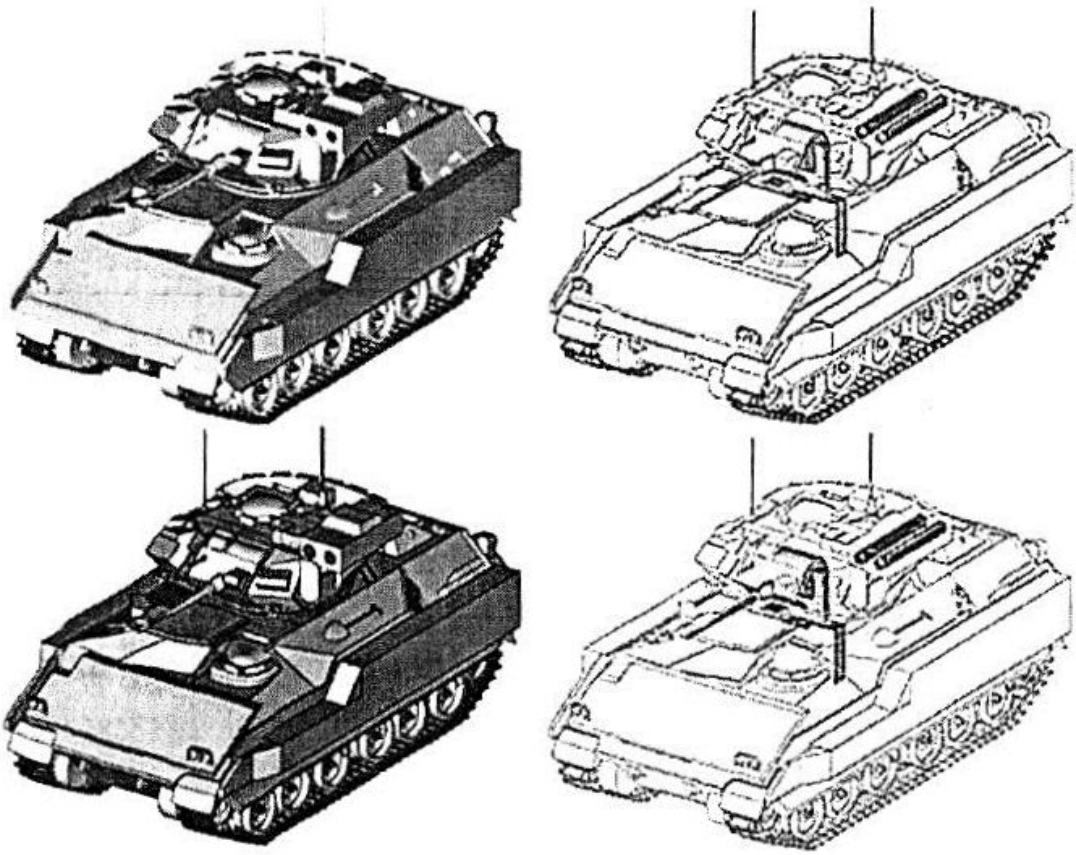


Hình 1.4. Một ảnh của não người được dựng lên với dữ liệu từ máy chụp cộng hưởng từ (MRI).



Hình 1.5. Từ rất nhiều lát chụp 2D, người ta có thể dựng lên thể hiện 3D của não.

Thiết kế hỗ trợ bởi máy tính (Computer-aided design). Trong thiết kế hỗ trợ bởi máy tính, người dùng tận dụng các công cụ đồ họa tương tác để thiết kế các thành phần và các hệ thống thiết bị cơ, điện, cơ điện, điện tử, thiết kế các cấu trúc như các tòa nhà, thân xe, thân máy bay và thân tàu thủy, v.v... Hình 1.6 cho thấy một ví dụ về thiết kế hỗ trợ bởi máy tính - đây là một bản thiết kế một chiếc xe tăng.

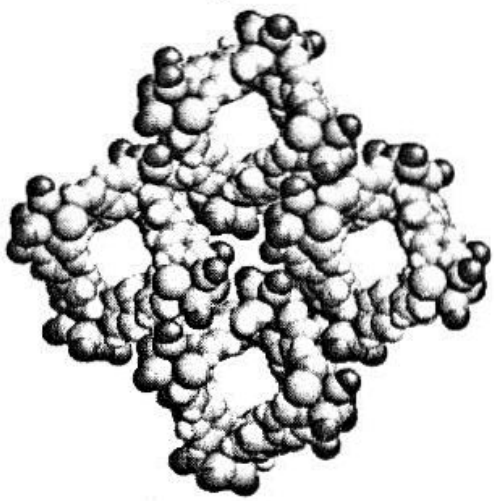


Hình 1.6. Bản vẽ thiết kế của một chiếc xe tăng.

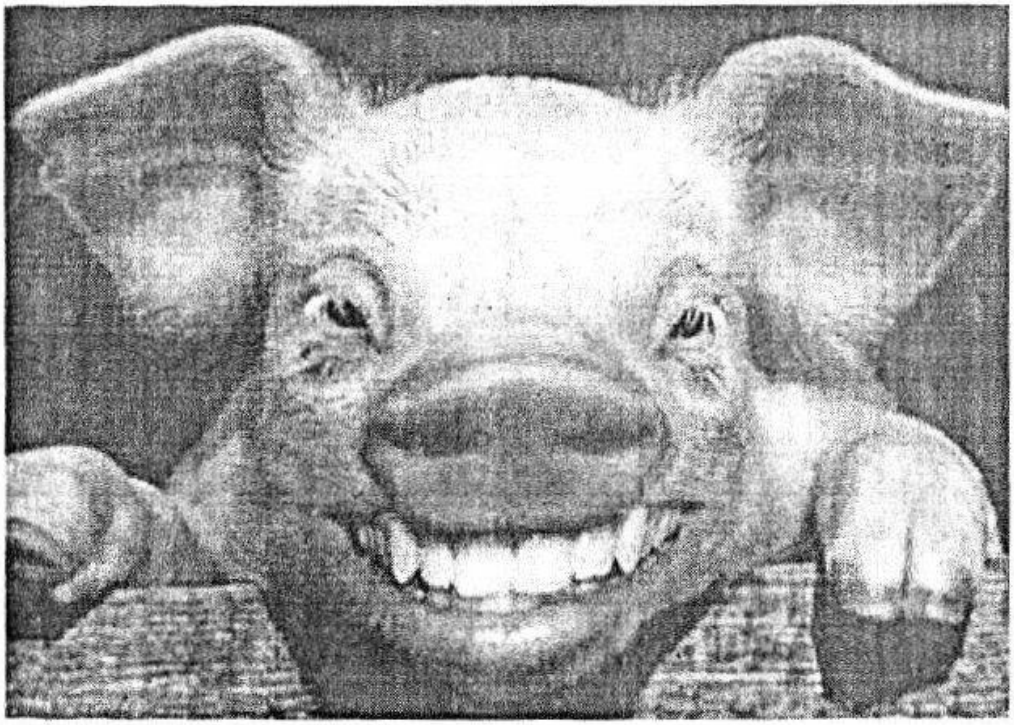
Các hệ thống đa phương tiện. Đồ họa máy tính đóng một vai trò cực kỳ quan trọng trong lĩnh vực đang phát triển – đa phương tiện Như hàm ý trong tên gọi, lĩnh vực này liên quan đến nhiều phương tiện liên lạc khác nhau, ví dụ như văn bản, đồ họa, âm thanh,... Hình 1.7 cho thấy những thành phần đa phương tiện có thể dùng trong giáo dục.

Hình 1.7. Hệ thống đa phương tiện có thể dùng trong giáo dục.

Mô phỏng và hoạt hình cho trực quan hóa khoa học (visualization) và giải trí. Các đoạn phim hoạt hình tạo ra bởi máy tính thể hiện các cách thức hoạt động của các vật thể thực và vật thể nhân tạo đang trở nên thông dụng trong việc trực quan hóa khoa học và kỹ thuật. Chúng ta có thể sử dụng chúng để nghiên cứu các đại lượng toán học trừu tượng cũng như các mô hình toán học của dòng chảy, thuyết tương đối, các phản ứng nguyên tử và hóa học, các hệ thống sinh lý học, và sự biến dạng của các cấu trúc cơ học dưới các lực tác động khác nhau (ví dụ Hình 1.8). Một lĩnh vực tiên tiến khác là tạo nên các kỹ xảo điện ảnh trong các bộ phim, ví dụ như trong Hình 1.9.



Hình 1.8. Sử dụng đồ họa máy tính
để mô phỏng sự hoạt động của các phân tử.



Hình 1.9. Đồ họa máy tính được sử dụng để tạo ra kỹ xảo điện ảnh.

1.1. Đôi chút về lịch sử đồ họa máy tính

Với đồ họa máy tính, điểm lại sự phát triển của phần cứng dễ hơn sự phát triển của phần mềm vì cuộc cách mạng phần cứng có ảnh hưởng lớn hơn nhiều tới sự phát triển của ngành đồ họa máy tính. Vì vậy, chúng ta sẽ bắt đầu với phần cứng.

Vào thời kỳ đầu của máy tính, đã có những thiết bị hiển thị lên giấy như Teletype và máy in dòng. Chiếc máy tính Whirlwind được phát triển năm 1950 tại MIT đã có màn hình hiển thị CRT cho đầu ra. Khởi điểm của đồ họa máy tính tương tác hiện đại bắt đầu từ luận văn tiến sỹ của Ivan Sutherland trên hệ thống vẽ Sketchpad. Ông đã giới thiệu những cấu trúc dữ liệu để lưu trữ cây phân cấp các biểu tượng thông qua tái tạo đơn giản các thành phần cơ bản, một kỹ thuật tương tự như việc dùng các mẫu plastic để vẽ các biểu tượng mạch. Sutherland cũng phát triển các kỹ thuật tương tác sử dụng bàn phím và bút quang (một thiết bị trỏ cầm tay có thể cảm nhận được ánh sáng phát ra từ các đối tượng trên màn hình) để lựa chọn, trỏ, và vẽ, và đã hình thành nhiều khái niệm và kỹ thuật nền tảng khác đang được dùng trong đồ họa máy tính hiện nay.

Cũng vào thời điểm đó, các nhà sản xuất máy tính, ô-tô và hàng không cũng nhận thấy rằng các hoạt động thiết kế và sản xuất hỗ trợ bởi máy tính (CAD/CAM) có một tiềm năng to lớn trong việc tự động hóa những khâu vẽ và làm mẫu tốn nhiều công sức. Hệ thống General Motors DAC (năm 1964) cho thiết kế ô tô và hệ thống Itek Digitek (năm 1981) cho thiết kế thấu kính là những chương trình tiên phong cho thấy việc tận dụng tương tác đồ họa trong các chu trình thiết kế tương tác rất phổ biến trong kỹ thuật. Đến giữa những năm 60 của thế kỷ 20, nhiều dự án nghiên cứu và sản phẩm thương mại về đồ họa máy tính bắt đầu xuất hiện.

Vì vào thời gian đó việc vào ra của máy tính chủ yếu dựa trên chế độ bó sử dụng các bìa đục lỗ, người ta hi vọng rất nhiều vào việc tạo ra một bước đột phá trong hình thức tương tác giữa con người và máy tính. Tuy vậy, phần cứng vẫn chưa cho phép đồ họa máy tính tạo ra nhiều kết quả. Thật vậy, đến đầu những năm 1980, đồ họa máy tính vẫn là một lĩnh vực nhỏ và chuyên biệt, chính bởi vì phần cứng còn rất đắt và các ứng dụng sử dụng đồ họa máy tính còn ít. Sau đó, các máy tính cá nhân với các thiết bị hiển thị đồ họa dạng mành, như Apple Macintosh và IBM PC và các máy nhái của nó, đã thông dụng hóa việc sử dụng đồ họa **ảnh nhị phân** (*bitmap*) cho tương tác giữa người sử dụng và máy tính. Một ảnh nhị phân có dạng là một ma trận chữ nhật các số không và một để thể hiện các điểm trên màn hình. Khi giá cả của các thiết bị đồ họa ảnh nhị phân bắt đầu chấp nhận được, đã có một sự bùng nổ của các ứng dụng rẻ và dễ sử dụng dựa trên đồ họa. Các giao diện người dùng dựa trên đồ họa đã cho phép hàng triệu người dùng mới có thể điều khiển các chương trình đơn giản và giá thành thấp, như bảng tính, soạn thảo văn bản và các chương trình vẽ.

Khái niệm **màn hình nền** (*desktop*) đã trở thành một phép ẩn dụ phổ biến về tổ chức không gian màn hình. Thông qua **trình quản lý cửa sổ** (*window manager*), người dùng có thể tạo ra, đặt vị trí, thay đổi kích thước của các vùng trên màn hình, được gọi là **cửa sổ** (*window*), hoạt động như các thiết bị đầu cuối đồ họa ảo chạy một ứng dụng nào đó. Cách tiếp cận này cho phép người dùng chuyển đổi giữa nhiều hoạt động chỉ bằng cách trỏ đến cửa sổ

mong muốn, thường với một thiết bị trỏ gọi là chuột (mouse). Giống như các mẩu giấy trên một cái bàn lộn xộn, các cửa sổ có thể đè lên nhau. Một phần của phép ẩn dụ về màn hình nền là sự hiện diện của các **biểu tượng** (*icon*), không chỉ thể hiện các tệp, thư mục hay chương trình ứng dụng, mà còn thể hiện các đối tượng thông dụng như hộp thư, máy in, thúng rác, có chức năng như các đối tượng trong cuộc sống.

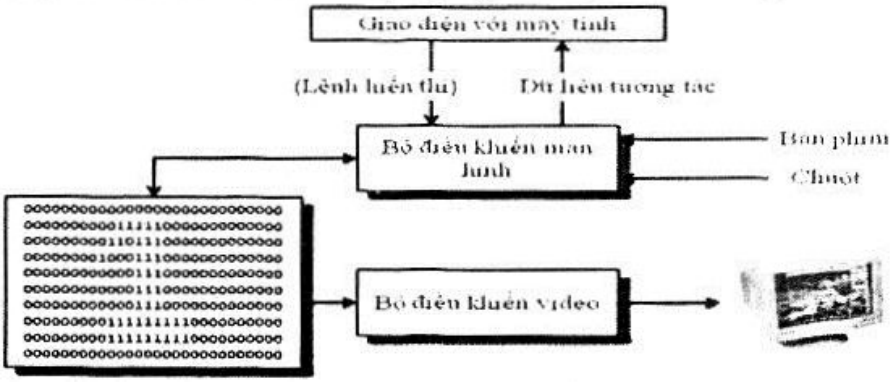
Thao tác trực tiếp với các đối tượng thông qua trỏ và bấm đã thay thế rất nhiều các câu lệnh được sử dụng trước đây. Như vậy, người sử dụng có thể chọn các biểu tượng hoặc chọn các nút hay các bảng chọn để kích hoạt các chương trình hay các đối tượng tương ứng. Ngày nay, các ứng dụng tương tác đã dùng đồ họa một cách tối đa cho giao diện người dùng và để trực quan hóa và thao tác với các đối tượng của ứng dụng.

1.2. Công nghệ hiển thị

Những thiết bị hiển thị thông dụng trong những năm giữa những năm 1960 và được dùng đến giữa những năm 1980 là các thiết bị hiển thị véc-tơ. Một hệ thống hiển thị véc-tơ thường gồm một bộ xử lý hiển thị kết nối với bộ xử lý trung tâm, một bộ đệm hiển thị, và một màn hình CRT. Với hệ thống thiết bị này, súng điện tử bắn từ một điểm đến một điểm khác trên lớp phốt-pho của màn hình CRT theo một lệnh hiển thị được đưa ra; kỹ thuật này được gọi là **quét ngẫu nhiên** (*random scan*). Vì ánh sáng tạo ra bởi lớp phốt-pho biến mất trong khoảng từ vài chục đến vài trăm micro giây, bộ xử lý hiển thị phải làm mới màn hình hiển thị bằng cách lặp qua danh sách hiển thị ít nhất là 30 lần một giây (30 Hz) để tránh bị giật.

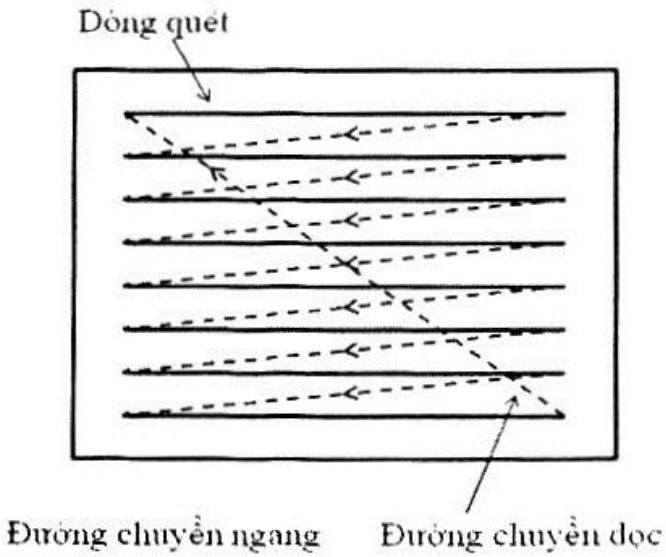
Sự phát triển của các thiết bị đồ họa mành với chi phí không cao dựa trên kỹ thuật ti-vi vào đầu những năm 1970 đã đóng góp lớn nhất cho sự phát triển của ngành đồ họa máy tính. Các thiết bị mành lưu trữ các đối tượng hiển thị cơ bản (như đường thẳng, ký tự, ...) trong một **bộ đệm làm mới** (*refresh buffer*) lưu trữ thành phần điểm của các đối tượng cơ bản, như trong Hình 1.10. Để có bức ảnh như mong muốn trên màn hình, chúng ta chỉ cần đặt đúng

các giá trị trong bộ đệm. Như vậy, chúng ta sẽ làm việc với các vật thể trong một không gian được gọi là **mành** (*raster*). Mành là một ma trận chữ nhật các **điểm ảnh** (*pixel*). Một dòng của mành được gọi là **đường quét** (*scanline*). Toàn bộ bức ảnh sẽ được một bộ điều khiển quét lần lượt từng dòng từ trên xuống dưới. Một súng điện tử sẽ bắn điện tử vào lớp phốt-pho để tạo nên cường độ sáng của điểm; trong các hệ thống màn hình màu, ba súng điện tử được dùng tương ứng với ba màu cơ bản đỏ, xanh lá cây và xanh nước biển. Ảnh hiển thị trên màn hình được tạo nên do sự kết hợp của các điểm được thắp sáng và không được thắp sáng. Súng điện tử bắt đầu quét từ góc trên của màn hình và quét từng dòng cho đến dưới đáy màn hình, sau đó lại quay trở lại lên trên (xem Hình 1.11).



Hình 1.10. Cấu trúc của một thiết bị hiển thị mành.

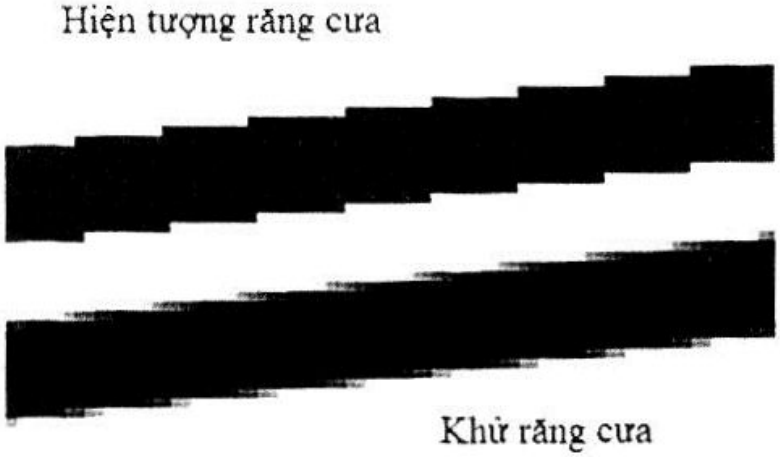
Lợi điểm chính của các thiết bị đồ họa mành đối với đồ họa véc-tơ là giá thành thấp hơn và khả năng tô phủ các vùng hiển thị với một màu hoặc một mẫu nhất định.



Hình 1.11.Màn hình dòng quét CRT.

Yếu điểm chính của các thiết bị mành đối với thiết vị véc-tơ là do sự rời rạc tạo ra do cách biểu diễn thông qua điểm. Trước hết, các đối tượng cơ bản như đường thẳng và đa giác được mô tả thông qua các điểm cuối/ đỉnh và sau đó phải được **quét chuyển** (*scan conversion*) thành danh sách các điểm trong bộ đệm khung. Việc quét chuyển này thường được thực hiện ở phần mềm chứ không phải ở phần cứng.

Một yếu điểm khác của các thiết bị mành nảy sinh từ bản chất của mành. Các hệ thống véc-tơ có thể vẽ các đường thẳng liên tục và trơn tru (thậm chí là các đường cong trơn tru) từ bất cứ điểm nào trên màn hình đến một điểm khác, các hệ thống mành chỉ có thể xấp xỉ các đường thẳng, đa giác, các đường cong bằng các điểm trên lưới mành. Sự xấp xỉ này có thể tạo ra các đoạn lởm chởm như trong Hình 1.12. Hiện tượng này là biểu hiện của một lỗi lấy mẫu được gọi là **răng cưa** (*aliasing*) trong lý thuyết xử lý tín hiệu; nó xuất hiện khi một hàm của một biến liên tục chứa những đoạn thay đổi đột ngột về cường độ được xấp xỉ bằng những mẫu rời rạc. Đồ họa máy tính hiện đại quan tâm đến những kỹ thuật loại bỏ răng cưa trên các hệ thống đa cấp xám hoặc các hệ thống màu (Hình 1.12). Các kỹ thuật này tạo ra sự thay đổi dần dần về cường độ tại các cạnh của các đối tượng cơ bản, thay vì chỉ đặt hai giá trị tối đa hoặc không.



Hình 1.12. Hiện tượng răng cưa và khử răng cưa với hệ thống đa cấp xám.

Một kỹ thuật hiển thị khác đã trở nên ngày càng thông dụng trong những năm gần đây là thiết bị hiển thị tinh thể lỏng – LCD (*liquid crystal display*). Mặc dù là một loại thiết bị hiển thị khác, LCD vẫn là các thiết bị dòng quét vì chúng vẫn gồm một ma trận chữ nhật các điểm, được làm mới từng dòng một. Như vậy, trong quyển sách này, chúng ta giả thiết thiết bị hiển thị trong tương lai gần là các thiết bị dòng quét. Đây là một giả thiết rất quan trọng và có ảnh hưởng lớn đến các thuật toán đồ họa máy tính hiện tại.

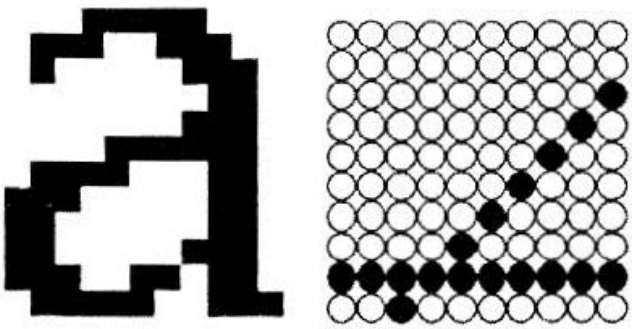
Câu hỏi và bài tập

1. Hãy kể thêm một số ứng dụng của Đồ họa máy tính mà chưa nêu trong chương này.
2. Tại sao các thuật toán đồ họa hiện tại lại dựa trên mành và đường quét?
3. Hiện tượng răng cưa là gì?

Chương 2
CÁC KHÁI NIỆM CƠ BẢN

2.1. Điểm ảnh

Đơn vị cơ bản xây dựng nên một bức ảnh trên màn hình máy tính, hay màn hình vô tuyến, là **điểm ảnh** (*pixel*). *Pixel* là viết tắt của "*picture element*" – thành phần ảnh. Một điểm ảnh có thể là hình tròn, hình vuông hoặc hình chữ nhật. Cũng giống như các ảnh được in trên giấy được tạo nên từ nhiều hàng, mỗi hàng gồm nhiều điểm ảnh, tạo ra cảm giác một ảnh liên tục nếu số lượng ảnh đủ lớn và được nhìn từ một khoảng cách nhất định. Các hàng điểm được quét qua màn hình bằng súng điện tử, và tập hợp các đường quét như vậy được gọi là mành.



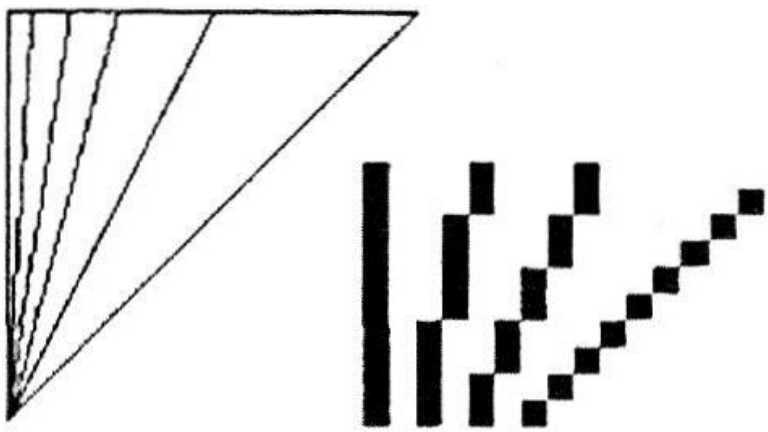
Hình 2.1. Một chữ cái được tạo nên từ các điểm ảnh hình vuông (trái) và hai đoạn thẳng giao nhau được tạo nên từ các điểm ảnh hình tròn.

Mật độ của các điểm ảnh xác định độ phân giải của ảnh. Càng nhiều điểm ảnh trên một ảnh, độ phân giải càng cao, và ảnh càng nét. Với màn hình đen trắng, một điểm được bật sáng thể hiện màu trắng, và tắt khi thể hiện màu đen. Một mẫu các điểm đen trắng, với tỉ lệ phù hợp sẽ tạo ra màu xám. Với màn hình đa cấp xám, độ sáng của mỗi điểm thể hiện cấp xám của điểm đó. Trên một màn hình

màu, mỗi điểm màu được định nghĩa bởi sự kết hợp của ba thành phần **đỏ** (*red*), **xanh lá cây** (*green*) và **xanh da trời** (*blue*) trong một hệ màu RGB. Chỉ đỏ, không xanh lá cây và không xanh da trời tạo ra một điểm màu đỏ. Có đủ cả ba thành phần tạo ra một điểm trắng. Một số hệ thống khác có thể sử dụng hệ màu HLS với ba thành phần: **màu sắc** (*hue*), **độ sáng** (*luminance*) và **độ tinh khiết** (*saturation*).

2.2. Mành và Véc-tơ

Khi một đối tượng được thể hiện thông qua một tập hợp các điểm trong một ma trận điểm 2 chiều, nó được gọi là ảnh mành. Khi đối tượng đó được thể hiện thông qua mối quan hệ trong không gian 2 chiều hay 3 chiều (ví dụ như đoạn thẳng, đoạn cong, v.v.), nó được gọi là ảnh véc-tơ. Ví dụ, một hình vuông dưới dạng ảnh véc-tơ là 4 đoạn thẳng từ (0,0) đến (0,100), (0,100) đến (100,100), (100,100) đến (100,0), và (100,0) đến (0,0). Hình vuông này dưới dạng ảnh mành là tập hợp các điểm (0,0), (0,1), ..., (0,100), v.v. Với các đoạn thẳng song song với trục tọa độ, các điểm này có thể được hiển thị một cách dễ dàng. Tuy nhiên, với các đoạn thẳng không song song với trục tọa độ, cần đến các thuật toán phức tạp hơn – các thuật toán vẽ đoạn thẳng (xem Hình 2.2).



Hình 2.2.Các đoạn thẳng trên màn hình được tạo nên từ tập hợp các điểm.

Newman [Newman84] đã đưa ra các tiêu chuẩn cho một đoạn thẳng trên thiết bị đầu ra:

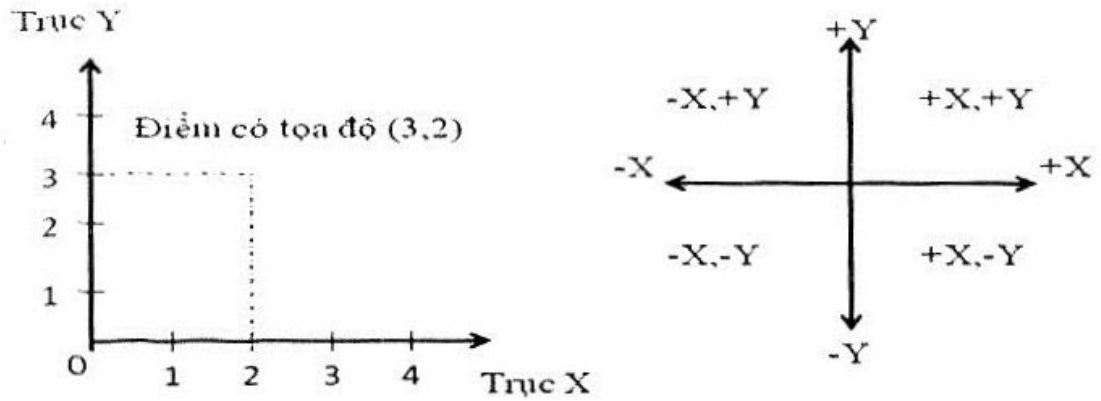
- Đoạn thẳng trông phải thẳng,

- Phải bắt đầu và kết thúc đúng điểm,
- Phải có mật độ điểm đều,
- Phải có mật độ điểm không phụ thuộc vào độ dài và hệ số góc của đoạn thẳng,
- Phải được vẽ ra một cách nhanh chóng.

Một trong những thuật toán vẽ đoạn thẳng được dùng rộng rãi là thuật toán Bresenham mà chúng ta sẽ đề cập trong Chương 3. Có thể nhận thấy rằng một đoạn thẳng được mô tả trong ảnh véc-tơ thì hoàn toàn chính xác, còn khi được mô tả trong ảnh mành thì độ chính xác phụ thuộc vào độ phân giải của thiết bị đầu ra (xem Hình 2.2).

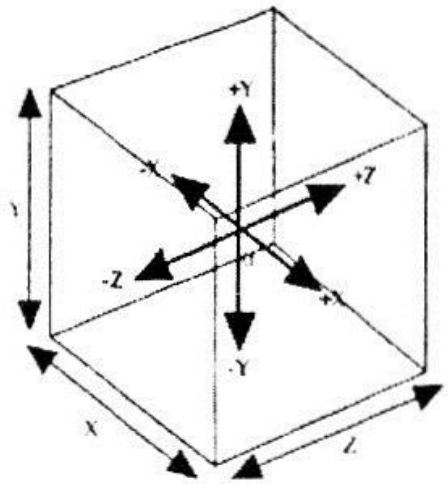
2.3. Tọa độ

Một điểm có thể được xác định thông qua hàng và cột của nó. Ví dụ, "cột 5 dòng 4" chỉ một điểm ở gần góc trái trên của màn hình và "cột 320,200" chỉ điểm trung tâm của một màn hình có kích thước 640x400. Tọa độ này được tính trong các hệ thống định nghĩa "0,0" là tọa độ góc trái trên của màn hình. Một số hệ thống khác định nghĩa "0,0" là tọa độ góc trái dưới của màn hình. Như vậy, một điểm trên một lưới 2 chiều có thể được xác định thông qua tọa độ Đề-các của nó (vị trí tương đối so với điểm gốc "0,0" với 2 trục **x** và trục **y** – Hình 2.3).



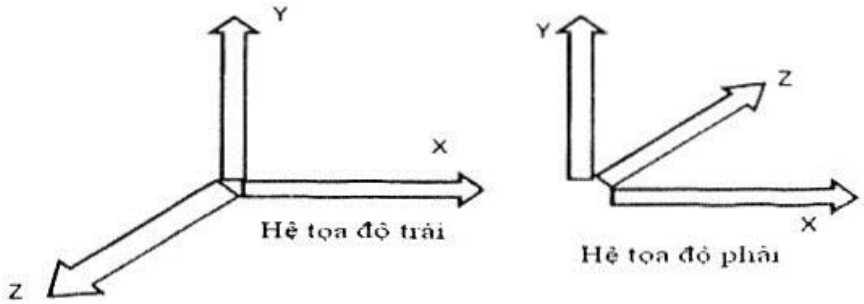
Hình 2.3. Hệ tọa độ Đề-các 2 chiều.

Để thuận tiện, người ta thường đặt gốc tọa độ tại trung tâm của màn hình, và như vậy ta sẽ có cả tọa độ âm và tọa độ dương. Trong không gian 3 chiều, ta cần thêm trục **z** vuông góc với mặt phẳng tạo bởi 2 trục **x** và **y** (Hình 2.4).



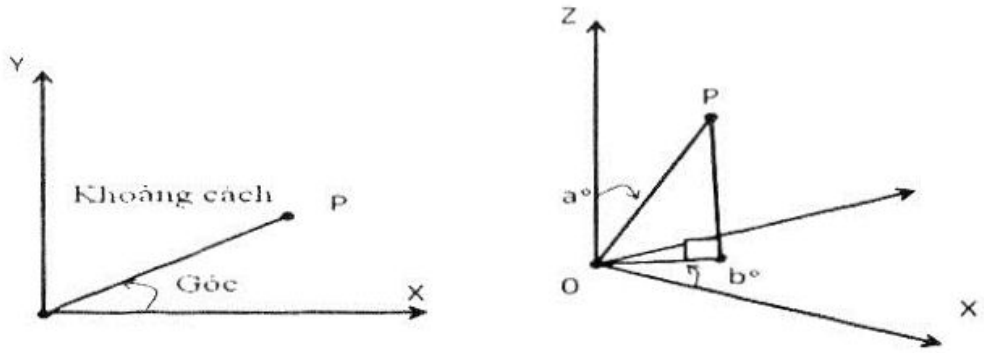
Hình 2.4. Hệ tọa độ Đề-các 3 chiều.

Hệ tọa độ Đề-các 3 chiều hơi phức tạp một chút vì với một số hệ thống theo **hệ tọa độ trái** (*left-handed*) - tọa độ **z** tăng khi điểm đi theo hướng ra xa khỏi người quan sát, và một số hệ thống lại là **hệ tọa độ phải** (*right-handed*) – tọa độ **z** tăng khi điểm đi theo hướng về người quan sát (Hình 2.5).



Hình 2.5. Hệ tọa độ trái và hệ tọa độ phải.

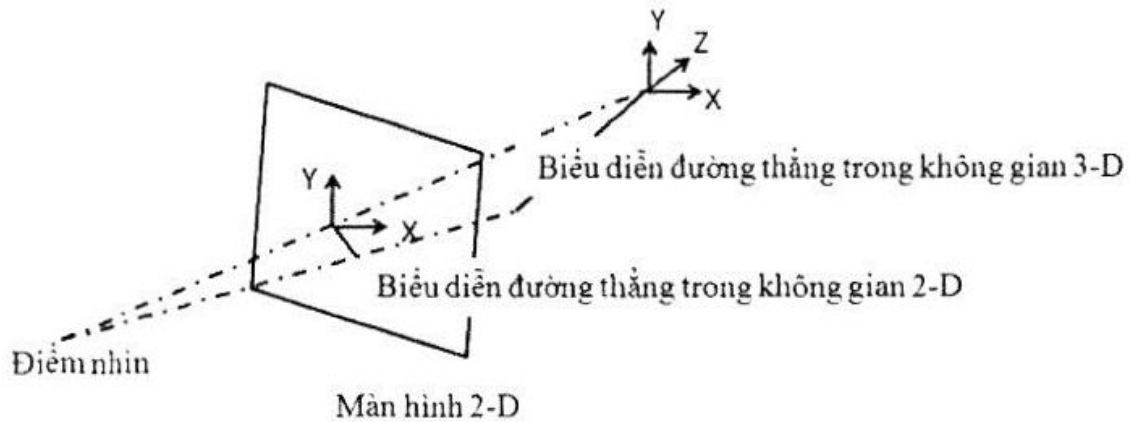
Một hệ tọa độ thông dụng khác là hệ tọa độ cực (2 chiều) và tọa độ cầu (3 chiều). Trong hệ tọa độ cực, một điểm được xác định bằng khoảng cách từ điểm đó đến gốc tọa độ và góc giữa trục dương x và tia từ gốc tọa độ đến điểm đó. Với tọa độ cầu, thì cần 2 góc thay vì chỉ một góc như tọa độ cực (Hình 2.6).



Hình 2.6. Hệ tọa độ cực và hệ tọa độ cầu.

Tọa độ dùng để xác định một đối tượng trong thế giới thực không nhất thiết phải bằng với tọa độ để xác định vị trí trên màn hình (hoặc bất cứ thiết bị ra nào). Thông thường, việc xác định các đối tượng được thực hiện thông qua một số hệ tọa độ quan hệ với nhau. Ví dụ, một con tem có vị trí là ở góc phải trên của phong bì, phong bì thì ở tâm của bàn, và bàn thì nằm cạnh tường của căn phòng. Khi cái bàn được di chuyển ra vị trí khác, chỉ vị trí tương đối giữa cái bàn và căn phòng phải xác định lại, còn các vị trí các vật trên cái bàn thì không thay đổi.

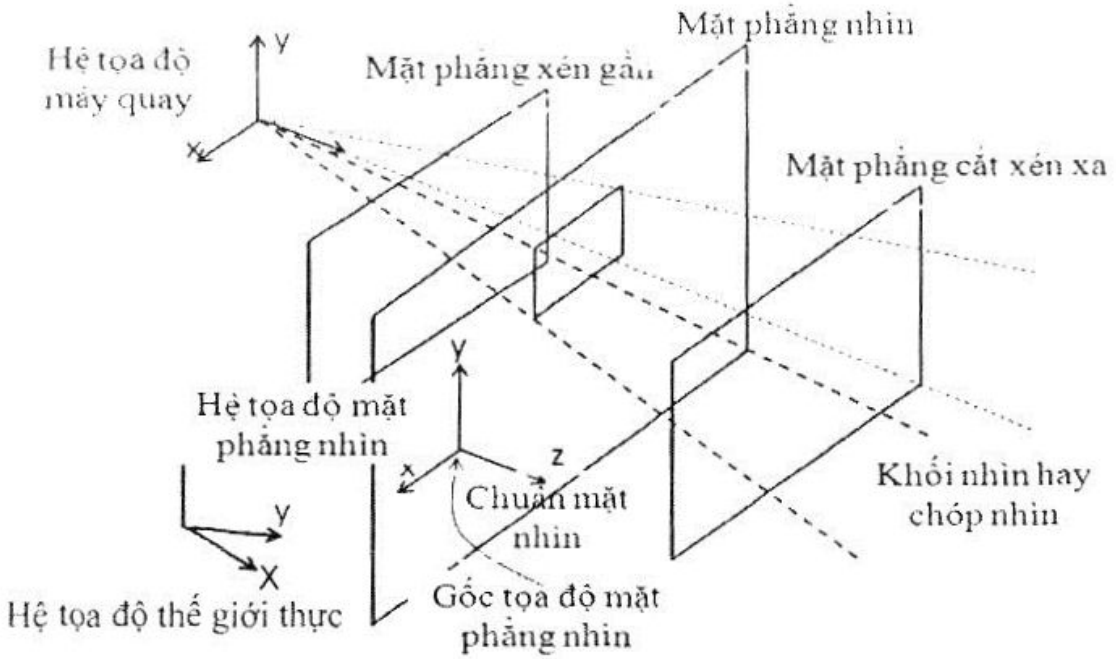
Vì màn hình là 2 chiều, các tọa độ trong một cảnh 3 chiều phải được chuyển về 2 chiều theo một cách nào đó. Thông thường, phép chiếu phối cảnh được sử dụng để làm việc này (Hình 2.7).



Hình 2.7. Chuyển từ tọa độ 3 chiều sang 2 chiều thông qua phép chiếu phối cảnh.

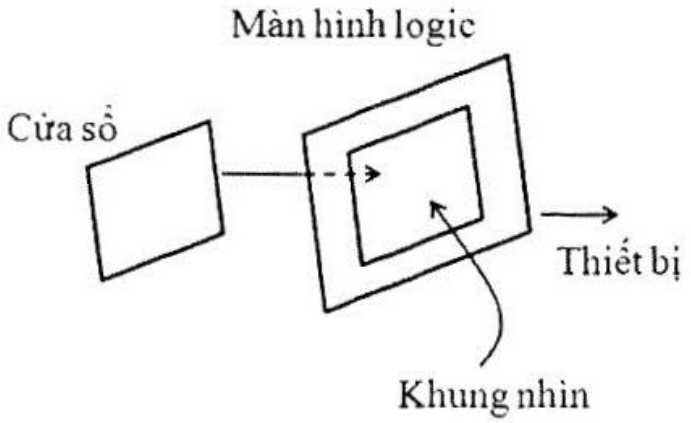
2.4. Luồng xử lý đồ họa

Để tạo được kết quả cuối cùng ở dạng ảnh trên thiết bị đầu ra, dữ liệu đầu vào là các đối tượng hình học để thể hiện cảnh vật được xử lý thông qua luồng xử lý đồ họa (*graphics pipepline*). Hình 2.8 cho thấy một số thuật ngữ trong các luồng xử lý đồ họa 3 chiều. Các đối tượng trong thế giới thực được mô tả bởi người sử dụng thông qua **hệ tọa độ thế giới thực** (*world coordinate system*). Cảnh vật thế giới được chiếu lên một **mặt phẳng nhìn** (*view plane*) từ một **điểm nhìn** (*viewpoint*) mà chúng ta thường biết đến là vị trí của **máy quay** (*camera*) hay mắt nhìn. Chúng ta cũng có **hệ tọa độ mặt phẳng nhìn** (*view plane coordinate system*) và **hệ tọa độ máy quay** (*camera coordinate system*) đi kèm. Hướng từ điểm nhìn dọc theo trục dương **z** của hệ tọa độ máy quay được định nghĩa là **hướng nhìn** (*view direction*). Một **cửa sổ** (*window*) trong mặt phẳng nhìn xác định vùng chữ nhật mà ta đang quan tâm. **Khối nhìn** (*view volume*) hay **chóp nhìn** (*view pyramid*) là một khối vô hạn tạo ra từ các tia xuất phát từ điểm nhìn đến các điểm trong cửa sổ. Để giới hạn các đối tượng đầu ra, người ta thường dùng **mặt phẳng cắt xén gần** (*near clipping plane*) và **mặt phẳng cắt xén xa** (*far clipping plane*). Khối nằm trong khối nhìn và chặn bởi hai mặt phẳng cắt xén được gọi là **khối nhìn đã cắt** (*truncated view volume*). Chỉ có các phần của đối tượng nằm trong khối này được chiếu lên cửa sổ và hiển thị. Thao tác tìm các phần này được gọi là **cắt xén** (*clipping*). Về cơ bản, ba hệ tọa độ - thế giới, máy quay và mặt phẳng nhìn – có thể độc lập. Tuy nhiên, trong thực tế, người ta thường giả thiết là các trục tọa độ của hệ tọa độ máy quay và mặt phẳng nhìn song song với nhau và tọa độ **z** vuông góc với mặt phẳng nhìn. Người ta cũng giả thiết là trục **x** và **y** của các hệ tọa độ này song song với các cạnh của cửa sổ.



Hình 2.8. Một số thuật ngữ và hệ tọa độ 3 chiều.

Bước cuối cùng trong việc chuyển một đối tượng đến thiết bị đầu ra liên quan đến việc biến đổi từ tọa độ mặt phẳng nhìn ra tọa độ của thiết bị đầu ra. Bước này thường có hai giai đoạn. Giai đoạn thứ nhất biến đổi cửa sổ thành một **khung nhìn** (*viewport*), là hình chữ nhật con của **màn hình logic** (*logical screen*). Ở giai đoạn thứ hai, tọa độ trên màn hình logic được biến đổi thành tọa độ của thiết bị đầu ra (xem Hình 2.9). Thỉnh thoảng, màn hình logic và thiết bị đầu ra có cùng hệ tọa độ, do đó, giai đoạn thứ hai không cần nữa.

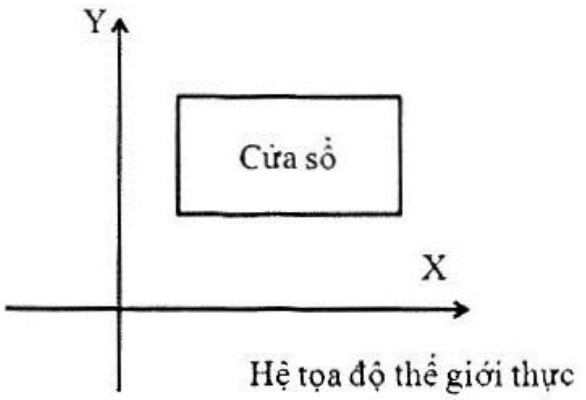


Hình 2.9. Các bước xử lý từ cửa sổ đến thiết bị.

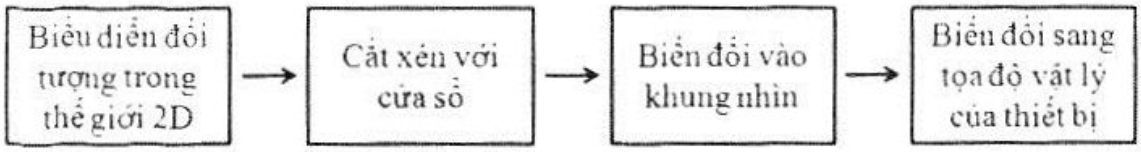
Các bước trong luồng xử lý đồ họa 3 chiều được tổng kết lại trong Hình 2.10.

Hình 2.10. Các bước trong luồng xử lý đồ họa 3 chiều.

Các bước trong luồng xử lý đồ họa 2 chiều cũng tương tự như 3 chiều. Các bước xử lý từ cửa sổ đến thiết bị như trong Hình 2.9 vẫn được giữ nguyên, nhưng Hình 2.8 và Hình 2.10 được thay bằng Hình 2.11 và Hình 2.12. Chúng ta có một hệ tọa độ thế giới 2 chiều và một cửa sổ với các cạnh song song với trục tọa độ. Trong các bước xử lý đồ họa 3 chiều, người ta thường giả thiết cửa sổ có kích thước cố định với tâm nằm trên trục Z của hệ tọa độ máy quay. Để thay đổi điểm nhìn và hướng nhìn, chúng ta chỉ cần thay đổi hệ tọa độ máy quay. Phóng to và thu nhỏ được thực hiện bằng cách di chuyển mặt phẳng nhìn gần vào hoặc xa ra khỏi điểm nhìn. Mặt khác, trong trường hợp các bước xử lý đồ họa 2 chiều, chúng ta phải cho phép cửa sổ này di chuyển và thay đổi kích thước. Chúng ta có thể di chuyển cửa sổ để nhìn thấy các phần khác nhau của thế giới và có thể kéo to hay thu nhỏ cửa sổ.



Hình 2.11. Hệ tọa độ đồ họa 2 chiều và cửa sổ.



Hình 2.12. Các bước trong luồng xử lý đồ họa 2 chiều.

Một điểm cần lưu ý là sự khác biệt giữa "cửa sổ" và "khung nhìn" trong các bước xử lý đồ họa chúng ta vừa trình bày. Trong một số trường hợp khác, người ta có thể dùng hai từ này thay thế cho nhau. Thực ra từ ngữ không quan trọng, mà quan trọng là khái niệm. Trong trường hợp này, chúng ta dùng từ "cửa sổ" để tập trung vào cái gì sẽ được nhìn thấy và từ "khung nhìn" để tập trung vào vị trí nào chúng ta sẽ quan sát.

Câu hỏi và bài tập

1. Luồng xử lý đồ họa là gì?
2. Hãy nêu các điều kiện cho một đường thẳng hoàn hảo được vẽ trên máy tính.
3. Ba màn hình có độ phân giải lần lượt là 640x480, 1024x768, 1280x1024. Hãy cho biết kích thước của bộ đệm màu nếu mỗi điểm ảnh được mô tả bằng 1 bit, 4 bit, và 8 bit.

Chương 3
CÁC THUẬT TOÁN MÀNH HÓA

Như đã giới thiệu trong Chương 2, một trong những yêu cầu đối với các thuật toán đồ họa máy tính chính là việc các đối tượng hình học phải được biểu diễn bằng một tập hợp các điểm trong một ảnh mành. Trong chương này, chúng ta sẽ làm quen với một số thuật toán chuyển các đối tượng từ ảnh véc-tơ sang ảnh mành, hay từ các đối tượng hình học liên tục thành tập các điểm trên lưới 2 chiều.

3.1. Các thuật toán tô phủ

Các thuật toán tô phủ các phần có viền bao được sử dụng khá rộng rãi. Với các phông chữ, người ta chỉ cần mô tả đường viền rồi sau đó mới tô phủ; với các phim hoạt hình, người họa sỹ vẽ đường viền trước rồi sau đó mới tô phủ, v.v. Có hai lớp thuật toán tô phủ: các thuật toán dựa trên đa giác (tô cạnh) và các thuật toán dựa trên điểm. Lớp thuật toán thứ nhất có thể sử dụng trong trường hợp phần cần tô được định nghĩa bởi đa giác và ta có thể dùng phương trình toán học để biểu diễn các cạnh. Lớp thuật toán thứ hai, tổng quát hơn, có thể được sử dụng khi đường viền được vừa có thể là đa giác vừa có thể là một tập các điểm.

Cũng cần lưu ý đến việc các thuật toán tô phủ xác định một điểm nằm trong khu vực cần tô như thế nào. Người ta có thể kiểm tra tính chẵn lẻ khi kẻ một tia từ điểm đó và đếm số lượng giao điểm với đường viền. Cách này luôn được dùng trong các thuật toán dựa trên đa giác và cũng được dùng với các thuật toán dựa trên điểm. Các thuật toán sử dụng *điểm hạt giống* thường dùng phương pháp dựa trên tính kết nối. Ở đây, bắt đầu từ điểm hạt giống, chúng ta có thể loang ra được các điểm xung quanh mà không vượt ra

ngoài đường viền bao. Cách này chỉ có thể áp dụng cho các thuật toán dựa trên điểm. Với cách này, chúng ta phải xác định điểm hạt giống nằm trong khu vực cần tô.

Bài toán tô phủ loang (*Flood fill problem*): *Với hai màu khác nhau* ***c*** *và* ***c'****, một tập các điểm* ***A*** *có cùng màu* ***c*** *được bao quanh bởi các điểm có màu khác với* ***c*** *và* ***c'****, tìm thuật toán thay màu của tất cả các điểm thuộc* ***A*** *và chỉ các điểm này thành màu* ***c'****.*

Thuật toán giải quyết bài toán tô phủ loang được gọi là thuật toán tô phủ loang. Ngoài ra, có một số bài toán và một số thuật toán liên quan đến tô phủ loang. Thuật toán 3.1.1 trình bày ý tưởng cơ bản của thuật toán tô phủ.

```
void BFA (int x, int y){
      if (Inside (x,y)){
            Set (x,y);
            BFA (x,y - 1); BFA (x,y + 1);
            BFA (x - 1,y); BFA (x + 1,y);
      }
}
```

Thuật toán 3.1.1. Thuật toán tô phủ cơ bản.

Trong Thuật toán 3.1.1, hàm **Inside**(x,y) xác định xem điểm (x,y) có nằm trong vùng cần tô. Thủ tục **Set**(x,y) đặt giá trị mong muốn cho điểm (x,y). Ví dụ, để có thuật toán tô phủ loang, ta chỉ cần cài đặt hàm **Inside**(x,y) để trả về giá trị True khi giá trị màu của điểm (x,y) giống với giá trị màu của điểm cần tô. Cần lưu ý là, Thuật toán 3.1.1 trông thì đơn giản, nhưng lại bùng nổ do có 4 lần gọi đệ quy. Nếu chỉ cài đặt như vậy thì thuật toán sẽ không hiệu quả vì mỗi điểm sẽ được đi nhiều lần. Nhiều cải tiến đã được đưa ra để tăng tính hiệu quả của thuật toán này. Smith [Smit79] đã đưa ra một thuật toán không đệ quy. Tuy chưa thực sự hiệu quả vì mỗi điểm vẫn phải đi qua 2 lần, thuật toán của Smith vẫn là nền tảng của nhiều thuật toán hiệu quả sau này.

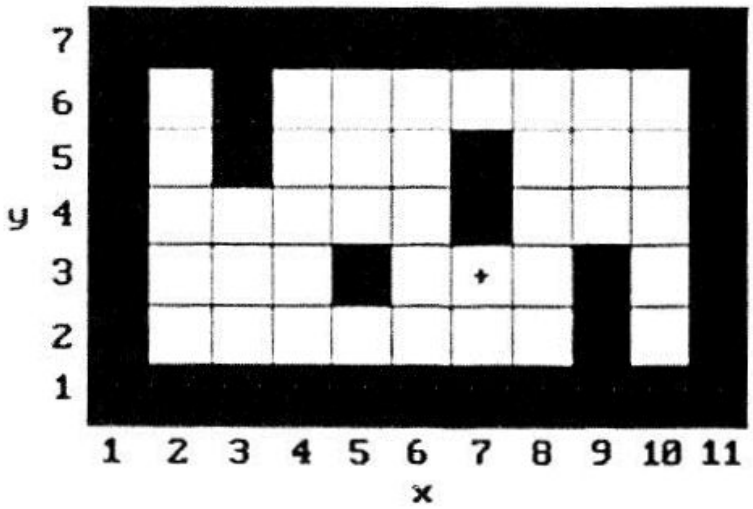
Trong thuật toán của Smith, các hằng số XMIN, XMAX, YMIN, và YMAX là các giá trị tối thiểu và tối đa cho tọa độ x và y của các điểm trong khung nhìn. Trong thuật toán này, một ngăn xếp được sử dụng với các hàm Push (đẩy vào ngăn xếp), Pop (lấy ra khỏi ngăn xếp), và StackNotEmpty (kiểm tra xem ngăn xếp có rỗng không).

```
/* Các biến toàn cục */
int x, y, lx, rx;
/*Một ngăn xếp chứa tọa độ các điểm (x,y);*/

void Fill (int seedx, int seedy){
      x = seedx; y = seedy;
      if (!(Inside (x,y)) exit(0);
      Push (x,y);
      while (StackNotEmpty()){
            PopXY ();
            if (Inside(x,y)){
               FillRight (); FillLeft ();
              /* Phủ các đoạn có chứa điểm */
               ScanHi (); ScanLo ();
              /*Quét các đoạn trên và dưới của
đoạn hiện tại*/
            }
      }
}

void FillRight (){
      int tx;
      tx =  x;
     /*Đi sang phải và thiết lập tất cả các điểm
của đoạn
      trong khi đi*/
      while (Inside (tx,y) && (tx <= XMAX)){
            Set (tx,y); tx = tx + 1;
      }
      rx =  tx - 1;
    /* Ghi lại chỉ số của điểm phải nhất trong
đoạn */
}

void FillLeft (){
      int tx;
```

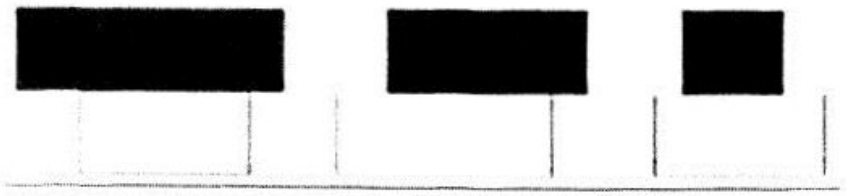
```
      tx =  x;
      /*Đi sang trái và thiết lập tất cả các điểm
của đoạn
      trong khi đi*/
      while (Inside (tx,y) && (tx >= XMIN)){
            Set (tx,y); tx =  tx - 1;
      }
      lx =  tx + 1;
      /* Ghi lại chỉ số của điểm trái nhất trong
đoạn */
}

void ScanHi ()
/*Quét qua các điểm giữa lx và rx trong dòng quét
trên dòng hiện tại. Cho vào ngăn xếp đoạn trái
nhất trong các đoạn mà ta tìm thấy. Không thiết
lập bất cứ điểm nào trong giai đoạn này.*/
{
      int tx;
      if (y + 1 > YMAX) exit(0);
      tx =  lx;
      while (tx <= rx){
            /* Bỏ qua những điểm không trong vùng
*/
            while (!(Inside (tx,y + 1)) && (tx <=
rx))
                  tx = tx + 1;
            if (tx <= rx){
            {
                  Push (tx,y + 1);
                  /*Chỉ lưu điểm đầu tiên của đoạn
ở trong
                  vùng. Sau đó quét qua tất cả các
điểm
                  trong đoạn. */
                  while (Inside (tx,y + 1) && (tx
<= rx))
                        tx = tx + 1;
            }
      }
}
```

Thuật toán 3.1.2. Thuật toán tô phủ loang Smith.

Ví dụ, trong Hình 3.1 ta bắt đầu từ điểm (7,3). Sau khi lệnh FillRight đầu tiên, đoạn (7,3) đến (8,3) được tô. Lệnh FillLeft sẽ tô điểm (6,3). Lệnh ScanHi sẽ cho lần lượt tọa độ điểm (6,4) và (8,4) vào ngăn xếp. Lệnh ScanLo sẽ thêm điểm (6,2). Các đoạn chứa (6,4), (8,4) và (6,2) được gọi là "vùng bóng tối" (*shadow*). Nhiệm vụ của ScanHi và ScanLo chính là tìm ra các "vùng bóng tối" để chuẩn bị tô. Sau đó chúng ta quay lại điểm bắt đầu của vòng lặp while chính, lấy (6,2) ra, và coi đây là điểm bắt đầu. Lệnh FillRight và FillLeft tiếp theo sẽ tô phủ đoạn từ (2,2) đến (8,2). ScanHi và ScanLo sẽ cho (2,3) và (6,3) vào ngăn xếp. Vòng lặp lại bắt đầu và lấy (6,3) ra. Lần này, (6,3) đã được tô nên chúng ta quay lại đầu vòng lặp và lấy ra (2,3) và cứ tiếp tục như thế cho đến khi ngăn xếp rỗng.



Hình 3.1. Ví dụ về thuật toán tô phủ.



Hình 3.2. Các loại vùng bóng tối.

Một vấn đề với thuật toán Smith là chúng ta phải xét mỗi điểm 2 lần, như điểm (2,3) trong ví dụ trên. Vấn đề này xảy ra chúng ta cho cả điểm ở trên và ở dưới của điểm hiện tại vào ngăn xếp. Sau đó, khi lấy điểm ở trên ra, ta lại phải xét lại điểm hiện tại. Chúng ta cần loại bỏ sự trùng lặp này để có một thuật toán nhanh hơn. Thuật toán 3.1.3 [Fish90] lưu trữ thêm một số thông tin để phân biệt ba loại "vùng bóng tối" khác nhau như trong Hình 3.2. Do vậy, thuật toán này chỉ xét trung bình mỗi điểm hơn một lần một chút và cũng thực hiện tốt kể cả trong trường hợp xấu nhất. Fishkin đã chỉ ra rằng, thuật toán này tối ưu nếu khu vực cần tô không có lỗ hổng ở giữa.

```
enum direction = {BELOW = -1, ABOVE = +1};
struct stackRec {
      /* Một bản ghi dữ liệu cho vùng bóng tối */
      int myLx, myRx;
      /* điểm kết thúc của vùng bóng tối này */
      int dadLx, dadRx; /* điểm kết thúc của vùng mẹ
*/
      int myY; /* dòng quét của vùng này */
      direction myDirection;
      /*-1 có nghĩa là ở dưới vùng mẹ, +1 có nghĩa
là ở
      trên */
};

/* Biến toàn cục */
stack of stackRec shadowStack;

void Fill (int seedx, int seedy){
      int x, lx, rx, dadLx, dadRx, y;
      direction dir;
      boolean wasIn;
      Khởi tạo shadowStack là rỗng;
      Tìm đoạn [lx,rx] chứa điểm hạt giống;
      Push (lx,rx,lx,rx,seedy+1,ABOVE);
      Push (lx,rx,lx,rx,seedy-1,BELOW);
      while (StackNotEmpty()){
            Pop ();
            if ((y < YMIN) || (y > YMAX)) continue;
            x = lx + 1;
```

```
            wasIn = Inside (lx, y);
            if (wasIn) {
                  Set (lx,y); lx =  lx - 1;
                  /* Nếu cạnh trái của vùng bóng tối
                   chạm vào một đoạn, di chuyển đầu
                   mút trái và thiết lập các điểm
                   trên đường đi */
                  while (Inside (lx,y) && lx >= XMIN)
{
                        Set (lx,y); lx =  lx - 1;
                  }
            }
            /*Bắt đầu vòng lặp chính. Đi sang phải
            bắt đầu từ vị trí x hiện tại, nếu wasIn
            là true, chúng ta ở trong một đoạn có
            cạnh trái tại lx.*/

            while (x <= XMAX){
                  if (wasIn){
                        if (Inside (x,y))
                              Set (x,y);
                          /*  đã ở trong và vẫn ở
                          trong */
                        else {
                              /* đã ở trong nhưng bây
                              giờ thì không, có nghĩa
                              là chúng ta vừa đi qua
                              cạnh phải của một đoạn*/
                           Stack(dadLx,dadRx,lx,x-1,
y, dir);
                              wasIn =  false;
                        }
                  }
                  else{
                        if (x <= rx){
                              if (Inside (x,y)){
                                    /*đã không ở trong
                                    nhưng bây giờ thì
                                    có, có nghĩa là
                                    chúng ta vừa gặp
                                    cạnh trái của một
                                    đoạn */
                                    Set (x,y);
                                    wasIn =  true;
                                    lx =  x;
```

```
                        }
                        x = x + 1;
                    }
                }
                if (wasIn){
                    /*chúng ta vừa chạm vào cạnh
                    của khung nhìn trong khi vẫn
                    ở trong một đoạn*/
                    Stack       (dadLx,dadRx,lx,x-
1,y,dir);
                }
            }
        }
}
bool StackNotEmpty ()
/* Trả về true nếu shadowStack là không rỗng và
false trong trường hợp ngược lại */

void Push (int myl, int myr, int dadl, int dadr,
int y, direction dir)
/* Đẩy bản ghi vào shadowStack */

void Pop ()
/* Lấy ra đỉnh của shadowStack vào các biến địa
phương lx, rx, dadLx, dadRx, y, dir */

void Stack (int dadLx, int dadRx, int lx, int rx,
int y, direction dir)
/* Đẩy thêm một vùng bóng tối vào shadowStack khi
biết đoạn vừa được phát hiện và mẹ của nó. Đây
chính là chỗ để phân biệt ba loại vùng bóng tối. */
{
    int pushrx, pushlx;
    pushrx =  rx + 1; pushlx =  lx - 1;
    Push (lx,rx,pushlx,pushrx,y+dir,dir);
    if (rx > dadRx)
        Push        (dadRx+1,rx,pushlx,pushrx,y-
dir,dir);
    if (lx < dadLx)
        Push        (lx,dadLx-1,pushlx,pushrx,y-
dir,dir);
}
```

Thuật toán 3.1.3. Thuật toán tô phủ Fishkin.

3.2. Các thuật toán vẽ đoạn thẳng

Bây giờ chúng ta sẽ bàn đến vấn đề thể hiện các đoạn thẳng bằng một tập các điểm rời rạc. Rõ ràng là chúng ta muốn ánh xạ cấu trúc liên tục của đoạn thẳng vào một cấu trúc rời rạc sao cho có thể giữ được các nhiều tính chất về hình dáng, như độ dày đồng đều và độ mịn của đoạn thẳng. Tuy nhiên đây không phải là công việc dễ dàng. Trong phần này, chúng ta sẽ tìm hiểu một số thuật toán vẽ đoạn thẳng.

3.2.1. Thuật toán vẽ đoạn thẳng DDA

Trước hết, chúng ta sẽ tìm hiểu thuật toán vẽ đoạn thẳng dựa trên phương trình vi phân. Thuật toán này được gọi là **phân tích số hóa vi phân** – *digital diferential analyzer* (DDA). Phương trình vi phân cho đường thẳng đi qua điểm (x_0,y_0) và (x_1,y_1) là:

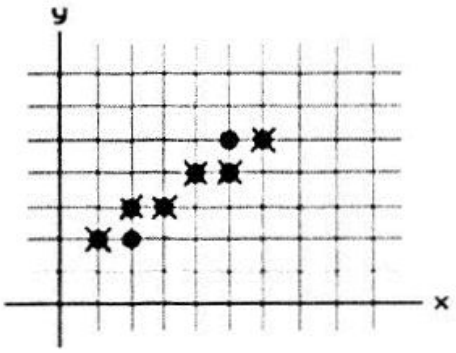
$$\frac{dx}{dy} = \frac{\Delta x}{\Delta y} = \frac{\varepsilon \Delta x}{\varepsilon \Delta y}, \tag{3.1}$$

với $\Delta y = y_1 - y_0$, $\Delta x = x_1 - x_0$, và e là một số thực dương. Phân tích số hóa phương trình này, một chuỗi điểm p_i được tạo ra:

$$p_{i+1} = p_i + (\varepsilon \Delta x, \varepsilon \Delta y) \tag{3.2}$$

Các điểm p_i này sẽ nằm trên đoạn thẳng từ (x_0,y_0) đến (x_1,y_1). Tuy nhiên, để có thể hiển thị trên thiết bị ra, ta cần các điểm q_i là các điểm với tọa độ làm tròn từ tọa độ của p_i. Như vậy q_i là một tập các điểm rời rạc với tọa độ nguyên để xấp xỉ đoạn thẳng liên tục từ (x_0,y_0) đến (x_1,y_1).

Đây là một thuật toán vẽ đoạn thẳng khá đơn giản. Tuy nhiên nó không hiệu quả vì liên quan nhiều đến các tính toán số thực. Tuy nhiên, nó chính là nền tảng cho thuật toán vẽ đoạn thẳng rất hiệu quả - thuật toán Bresenham.



× DDA đơn giản • DDA đối xứng

Hình 3.3. Các đoạn thẳng sinh ra bởi thuật toán DDA đơn giản và DDA đối xứng.

Một điểm cần lưu ý là giá trị ε được chọn như thế nào rất ảnh hưởng đến kết quả của thuật toán. Chúng ta sẽ xem hai cách lựa chọn ε. Hai cách lựa chọn này tạo ra hai thuật toán DDA khác nhau: **thuật toán DDA đơn giản** (*simple DDA*) và **thuật toán DDA đối xứng** (*symmetric DDA*).

Đặt $m = \max(|\Delta x|, |\Delta y|)$.

DDA đơn giản: Chọn $\varepsilon = 1/m$

DDA đối xứng: Chọn $\varepsilon = 2^{-n}$ với $2^{n-1} \le m < 2^n$

Ví dụ: Ta muốn vẽ đoạn thẳng từ (1,2) đến (6,5).

Trong trường hợp này, $\Delta x = 5$ và $\Delta y = 3$. Với DDA đơn giản, chúng ta có:

$$\varepsilon = 1/5, \varepsilon\Delta x = 1, \varepsilon\Delta y = 3/5, \text{ và } p_{i+1} = p_i + (1, 3/5)$$

Với DDA đối xứng, chúng ta có:

$$\varepsilon = 1/8, \varepsilon\Delta x = 5/8, \varepsilon\Delta y = 3/8, \text{ và } p_{i+1} = p_i + (5/8, 3/8)$$

Các điểm được sinh ra có thể xem trong Hình 3.3. Các điểm do DDA đơn giản sinh ra được thể hiện bằng các dấu x, và các điểm do DDA đối xứng sinh ra được thể hiện bằng các chấm tròn.

3.2.2. Thuật toán vẽ đoạn thẳng Bresenham

Mặc dù thuật toán vẽ đoạn thẳng DDA khá đơn giản, nó lại liên quan đến các tính toán số thực. Một số cải tiến giúp cho thuật toán chỉ liên quan đến số nguyên, mà thực chất là chỉ liên quan đến phép cộng/trừ.

Sau đây là phần mô tả thuật toán vẽ đoạn thẳng Bresenham [Bres65]. Thuật toán này, hoặc biến thể của nó, thường được dùng để vẽ đoạn thẳng trong máy tính. Brensenham đã chứng minh trong [Bres77] rằng thuật toán của ông tạo ra xấp xỉ rời rạc tốt nhất của một đoạn thẳng liên tục.

Lưu ý rằng, trong trường hợp thuật toán DDA đơn giản, hoặc x hoặc y sẽ luôn luôn được tăng lên 1. Để đơn giản hóa, giả thiết điểm xuất phát của đoạn thẳng là ở gốc tọa độ. Nếu chúng ta giả thiết thêm rằng đoạn thẳng chúng ta cần vẽ ở góc phần tám thứ nhất của mặt phẳng, thì x luôn luôn được tăng lên 1. Do đó, chúng ta chỉ cần quan tâm đến việc tính tọa độ y cho hiệu quả.

Giả thiết đoạn thẳng chúng ta cần vẽ là từ (0, 0) đến (a, b), với a và b là 2 số nguyên, và $0 \le b \le a$ (vì (a, b) ở góc phần tám thứ nhất). Sử dụng phương trình vi phân như trình bày trong mục 3.2.1, điểm p_i, $0 \le i \le a$, sinh ra bởi thuật toán DDA đơn giản được xác định như sau:

$$p_i = p_{i-1} + (1, \frac{b}{a}) = (i, i \times \frac{b}{a}) \qquad \textbf{(3.3)}$$

và biểu diễn của đoạn thẳng sẽ là tập các điểm (i, y_i), với y_i là làm tròn của số thực $i \times \frac{b}{a}$.

Giá trị của tọa độ y bắt đầu từ 0. Câu hỏi đặt ra là "tại điểm nào, y_i sẽ bắt đầu bằng 1?". Để trả lời câu hỏi này, chúng ta phải tính $\frac{b}{a}$, $\frac{2b}{a}$, $\frac{3b}{a}$, ..., và xem tại điểm nào các giá trị này bắt đầu lớn hơn $\frac{1}{2}$. Thêm vào đó, giá trị y_i sẽ vẫn giữ bằng 1 cho đến khi các giá trị đó lớn $\frac{3}{2}$. Lúc đó, giá trị của y_i bắt đầu bằng 2. Như vậy

chúng ta phải so sánh $\frac{b}{a}, \frac{2b}{a}, \frac{3b}{a}, \ldots$ với các số $\frac{1}{2}, \frac{3}{2}, \frac{5}{2}, \ldots$ Vì chúng ta muốn tránh làm các phép tính số thực, chúng ta sẽ thay bằng các phép so sánh *2b*, *4b*, *6b*, ... với *a*, *3a*, *5a*, ... Vì việc so sánh một số với 0 nhanh hơn việc so sánh 2 số với nhau, chúng ta sẽ bắt đầu với việc xét khi nào thì *2b-a*, *4b-a*, ... bắt đầu lớn hơn 0. Đặt giá trị ban đầu $d = 2b-a$, sau đó mỗi lần chỉ cần cộng thêm *2b* vào. Khi *d* bắt đầu lớn hơn 0, thì ta bắt đầu phải kiểm tra xem *d* có lớn hơn *2a* hay không. Tiếp tục lấy *d* trừ đi *2a*, ta lại chuyển về phép so sánh có lớn hơn 0 hay không. Như vậy, chúng ta luôn luôn chỉ phải kiểm tra xem *d* có lớn hơn 0 hay không.

Giả sử, chúng ta cần vẽ một đoạn thẳng từ (0,0) đến (15,3). Trong trường hợp này, $2a = 30$, $2b = 6$, và giá trị khởi điểm của *d* là $6 - 15 = -9$. Bảng sau cho thấy các điểm (x_i, y_i) được vẽ và sự thay đổi của *d* với *i* từ 0 đến 8.

i	0	1	2	3	4	5	6	7	8
d		−9	−3	−27	−21	−15	−9	−3	−27
(x_i,y_i)	(0,0)	(1,0)	(2,0)	(3,1)	(4,1)	(5,1)	(6,1)	(7,1)	(8,2)

Đoạn mã sau là cài đặt của thuật toán mà chúng ta vừa mô tả ở trên để vẽ đoạn thẳng từ (0, 0) đến (*a*, *b*) ở góc phần tám thứ nhất:

```
{
      int d, x, y;
      d =  2*b - a;
      x =  0;
      y =  0;

      while (true){
            Draw (x,y);
            if (x == a) exit(0);
            if (d >= 0){
                  y =  y + 1;
                  d =  d - 2*a;
            }
            x =  x + 1;
            d =  d + 2*b;
      }
}
```

Trong phần trên, chúng ta mới chỉ đề cập đến các đoạn thẳng bắt đầu từ gốc tọa độ đến điểm nằm trong góc phần tám thứ nhất. Muốn vẽ đoạn thẳng bắt đầu từ một điểm khác, chúng ta chỉ việc cộng thêm phần khác biệt vào tất cả các điểm cần vẽ. Việc xử lý những trường hợp khi điểm cuối của đoạn thẳng không nằm ở góc phần tám thứ nhất cũng tương tự, bằng cách đổi chỗ x cho y và đổi dấu. Sau đây là một biến thể của thuật toán vẽ đoạn thẳng của Bresenham được đề xuất trong [Heck90] và vẽ được đoạn thẳng trong mọi trường hợp.

```
void DrawLine (int x0, int y0, int x1, int y1){
      int dx, ax, sgnx, dy, ay, sgny, x, y, d;

      dx =  x1 - x0; ax =  abs (dx)*2; sgnx =
Sign (dx);
      dy =  y1 - y0; ay =  abs (dy)*2; sgny =
Sign (dy);
      x =  x0; y =  y0;

      if (ax > ay){ /* x tăng nhanh hơn y */
            d =  ay - ax/2;
            while (true){
                  Draw (x,y);
                  if (x = x1) exit(0);
                  if (d >= 0){
                        y =  y + sgny; d =  d - ax;
                  }
                  x =  x + sgnx; d =  d + ay;
            }
      }
      else{ /* y tăng nhanh hơn x */
            d =  ax - ay/2;
            while (true){
                  Draw (x,y);
                  if (y == y1) exit(0);
                  if (d >= 0){
                        x =  x + sgnx; d =  d - ay;
                  }
                  y =  y + sgny; d =  d + ax;
            }
      }
}
```

```
int Sign (double x){
      if (x < 0) return (-1)
      else return (+1);
}
```

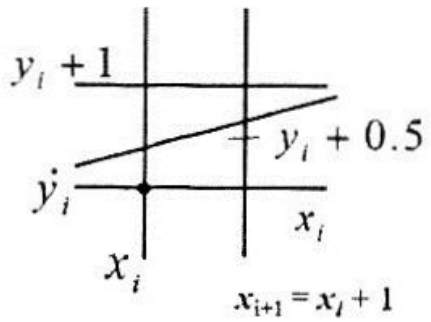
3.2.3. Thuật toán vẽ đoạn thẳng điểm giữa

Vì việc vẽ đoạn thẳng hiệu quả rất quan trọng trong đồ họa máy tính, người ta luôn tìm cách phát triển ra các thuật toán tốt hơn. Một thuật toán vẽ đoạn thẳng nổi tiếng khác được gọi là thuật toán vẽ đoạn thẳng điểm giữa (*Midpoint Line-Drawing Algorithm*). Nó cũng tạo ra các điểm như thuật toán Bresenham, nhưng lại phù hợp hơn cho các đường cong và đoạn thẳng trong không gian ba chiều. Ý tưởng cơ bản được mô tả đầu tiên trong [Pitt67] và sau đó được mô tả chi tiết trong [VanN85].

Giả thiết L là một đoạn thẳng không thẳng đứng được định nghĩa bởi phương trình

$$f(x,y) = ax + by + c = 0, \qquad (3.4)$$

với $0 \le a \le -b$. Giả thiết này có nghĩa là đoạn thẳng này có hệ số góc từ 0 đến 1. Các hệ số góc khác cũng được xử lý như trong thuật toán Bresenham. Đoạn thẳng thẳng đứng được xử lý riêng. Một hệ quả quan trọng của giả thiết trên chính là $f(x,y)$ lớn hơn 0 với những điểm nằm trên đoạn thẳng đó, và $f(x,y)$ nhỏ hơn 0 với những điểm nằm dưới đoạn thẳng đó. Cũng giống như thuật toán Bresenham, các điểm $p_i = (x_i,y_i)$ được sinh ra cũng có tính chất là tọa độ x được tăng lên 1 mỗi lần, $x_{i+1} = x_i + 1$, do đó chúng ta chỉ cần xác định giá trị của y_i.



Hình 3.4. Đại lượng quyết định trong thuật toán vẽ đoạn thẳng điểm giữa.

Xem trong Hình 3.4, giá trị của y_{i+1} chỉ có thể là y_i hoặc $y_i + 1$, và phụ thuộc vào dấu của:

$$d_i = f(x_i + 1, y_i + 0.5) \tag{3.5}$$

Nếu $d_i > 0$, đoạn thẳng L sẽ cắt đoạn thẳng $x = x_i + 1$ tại điểm nằm trên điểm $(x_i + 1, y_i + 0.5)$, do đó giá trị của y_{i+1} sẽ là $y_i + 1$. Nếu $d_i < 0$, giá trị của $y_i + 1$ sẽ là y_i. Nếu $d_i = 0$, một trong hai giá trị y_i và $y_i + 1$ đều được. Trong trường hợp này, ta sẽ chọn y_i. Đây chính là ý tưởng cơ bản của thuật toán điểm giữa. Việc tiếp theo là phải hiệu quả hóa việc tính toán. Trước hết, d_i cần được tính toán một cách hiệu quả một cách tăng dần. Như vậy, nếu $d_i \leq 0$, thì

$$\begin{aligned} d_{i+1} &= f(x_i + 2, y_i + 0.5) \\ &= a(x_i + 2) + b(y_i + 0.5) + c \\ &= d_i + a \end{aligned} \tag{3.6}$$

Ngược lại, nếu $d_i > 0$, thì

$$\begin{aligned} d_{i+1} &= f(x_i + 2, y_i + 1.5) \\ &= a(x_i + 2) + b(y_i + 1.5) + c \\ &= d_i + a + b \end{aligned} \tag{3.7}$$

Do đó, các giá trị tiếp theo của biến quyết định d_i có thể được tính toán một cách đơn giản bằng cách cộng thêm vào giá trị trước đó một lượng nhất định. Nếu đoạn thẳng xuất phát từ điểm $p_0 = (x_0, y_0)$, thì

$$d_0 = f(x_0 + 1, y_0 + 0.5) = f(x_0, y_0) + a + b/2 \tag{3.8}$$

Đây chính là giá trị khởi đầu của biến quyết định, và các giá trị tiếp theo sẽ được tính toán một cách tăng dần. Vì các so sánh đều với 0, ta có thể tránh phép tính chia 2 ở đây bằng cách nhân tất cả các giá trị quyết định với 2. Có nghĩa là, giá trị khởi đầu của d_0 có thể thay bằng

$$d_0 = 2f(x_0,y_0) + 2a + b = 2a + b \qquad (3.9)$$

và giá trị tăng thêm cũng được nhân với 2. Sau đây là cài đặt của thuật toán điểm giữa:

```
void DrawLine (int x0, int y0, int x1, int y1){
      int dx, dy, d, posInc, negInc, x, y;
      dx =  x1 - x0;
      dy =  y1 - y0;
      d =  2*dy - dx;
      /*Giá trị khởi điểm cho biến quyết định*/
      posInc =  2*dy;
      /*Giá trị tăng thêm cho d khi d >= 0*/
      negInc =  2*(dy - dx);
      /*Giá trị tăng thêm cho d khi d < 0*/
      x =  x0; y =  y0;
      Draw (x,y);
      while (x < x1){
            if (d <= 0)
                  d =  d + posInc;
            else{
                  d =  d + negInc; y =  y + 1;
            }
            x =  x + 1;
            Draw (x,y);
      }
}
```

Cần lưu ý là phần lớn thời gian trong việc vẽ đoạn thẳng được sử dụng bởi thao tác đặt giá trị cho các điểm trong bộ đệm, do đó, chúng ta không chỉ cần cải tiến thuật toán vẽ trong phần mềm mà còn cần các cải tiến về mặt phần cứng.

Câu hỏi và bài tập

1. Xác định các điểm sinh ra do dùng thuật toán DDA đơn giản và thuật toán DDA đối xứng để vẽ đoạn thẳng (2, 6) đến (-1, 1).
2. Hãy nêu ý nghĩa của việc nhân các đại lượng so sánh với 2 trong thuật toán vẽ đoạn thẳng Bresenham.
3. Trong thuật toán vẽ đoạn thẳng Bresenham, tại sao người ta lại so sánh 2b-a với 0 mà không so sánh 2b với a?
4. **Bài tập lập trình:** Chương trình vẽ - hãy viết một chương trình nhập vào một đa giác (số lượng đỉnh và tọa độ các đỉnh), sau đó cài đặt thuật toán vẽ đoạn thẳng Bresenham để vẽ các cạnh của đa giác và cài đặt thuật toán tô phủ Fishkin để tô màu đa giác.

Chương 4
CÁC THUẬT TOÁN CẮT XÉN

Các thuật toán cắt xén (clipping) được đánh giá là loại thuật toán quan trọng thứ hai trong đồ họa máy tính, chỉ sau các thuật toán vẽ đoạn thẳng. Về mặt toán học, cắt một tập bằng một tập khác có nghĩa là tìm phần giao của chúng. Khi cài đặt, người ta cũng muốn tìm phần giao này theo một cấu trúc dữ liệu định trước nào đó. Chương này sẽ thảo luận một số thuật toán cắt xén thông dụng cùng với một số thuật toán mới và hiệu quả hơn. Các thuật toán cắt xén được chia làm hai loại: cắt xén đoạn thẳng - thực hiện việc cắt các đoạn thẳng bằng một hình chữ nhật hay một đa giác lồi; và cắt xén đa giác - thực hiện cắt toàn bộ đa giác bằng một đa giác khác.

Định nghĩa: Đa giác được cắt gọi là ***"đa giác đối tượng"*** (*subject polygon*) và đa giác được dùng để cắt được gọi là ***"đa giác cắt"*** (*clip polygon*).

Các thuật toán được chọn để trình bày trong chương này dựa trên một trong các tiêu chí sau:

(1) Thuật toán đó là một trong những thuật toán tốt nhất của một loại nào đó tại thời điểm hiện tại;

(2) Thuật toán đó không phải thuật toán tốt nhất nhưng vẫn được dùng nhiều;

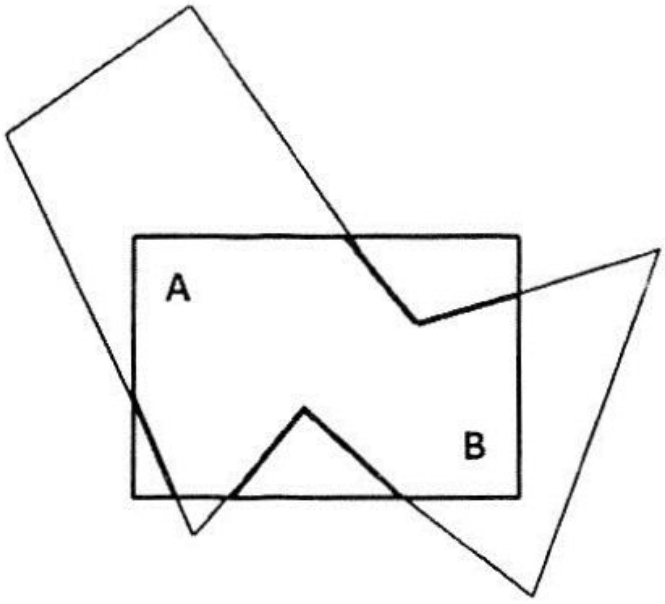
(3) Thuật toán đó có ý nghĩa về mặt lịch sử và dễ mô tả;

(4) Thuật toán đó liên quan đến các kỹ thuật đáng chú ý, mặc dù nó không còn được sử dụng nữa.

Các thuật toán cắt xén đoạn thẳng cũng được chia làm hai loại: Loại dùng cách mã hóa các đầu của đoạn thẳng (Cohen-Sutherland)

và loại dùng phương trình tham số để xác định các đoạn thẳng (Cyrus-Beck, Liang-Basky, và Nicholl-Lee-Nicholl).

Thông thường, người ta thường cần cắt nhiều hơn một cạnh tại một thời điểm, ví dụ như khi muốn cắt một đa giác bằng một đa giác khác. Người ta có thể thực hiện lần lược việc cắt từng đoạn thẳng. Tuy nhiên đây không phải là cách hiệu quả nhất vì nó cần thêm bộ nhớ để lưu trữ khi có những điểm mới xuất hiện trong quá trình cắt xén. Ví dụ trong Hình 4.1, hai góc A và B của cửa sổ cắt sẽ được thêm vào phần kết quả. Những góc này được gọi là **điểm rẽ** (*turning point*). Khái niệm này được giới thiệu trong [LiaB83] và dùng để chỉ những điểm nằm trên phần các cạnh giao của 2 vùng cắt mà cần phải được thêm vào để giữ tính liên tục của đa giác ban đầu. Chính vì vậy bài toán cắt đa giác thường được xem xét độc lập với bài toán cắt đoạn thẳng.



Hình 4.1. Cắt đa giác bằng một hình chữ nhật.

Các thuật toán cắt xén đa giác thường được chia làm hai loại: các thuật toán dựa trên điểm rẽ, như thuật toán Liang-Barsky và Maillot, được thực hiện bằng cách nhanh chóng tìm ra các điểm rẽ một cách rõ ràng, và loại còn lại. Loại dựa trên điểm rẽ quét qua các đoạn của *đa giác đối tượng* và cắt mỗi đoạn đó. Các thuật toán còn lại tìm các điểm rẽ một cách không chủ ý, có nghĩa là các thuật toán không chủ đích tìm kiếm các điểm này, mà chúng sẽ

được sinh ra một cách "tự động" khi thuật toán được thực hiện. Thuật toán Sutherland-Hodgman coi *đa giác cắt* như là phần giao của các nửa mặt phẳng, và do đó, *cắt đa giác đối tượng* bằng các nửa mặt phẳng này. Các thuật toán Weiler, Vatti, và Greiner-Hormann tìm ra các điểm rẽ từ *đa giác cắt* trong quá trình dò theo đường viền của phần giao đa giác. Mỗi thuật toán dò đường viền theo một cách khác nhau.

4.1. Các thuật toán Cắt xén đoạn thẳng

4.1.1. Thuật toán Cắt xén đoạn thẳng Cohen-Sutherland

Bài toán cắt xén đoạn thẳng trên mặt phẳng được mô tả như sau:

> Cho một đoạn [P1,P2], phải cắt nó với một cửa sổ hình chữ nhật và trả về đoạn được cắt [Q1,Q2] hoặc trả về rỗng nếu đoạn [P1,P2] nằm hoàn toàn ngoài cửa sổ cắt.

Thuật toán cắt đoạn thẳng Cohen-Sutherland có lẽ là thuật toán phổ biến nhất bởi sự đơn giản của nó. Nó bắt đầu bằng việc mã hóa chín khu vực phân chia bởi đường thẳng chứa các cạnh của cửa sổ bằng mã 4 bit (xem Hình 4.2). Nếu P là một điểm bất kỳ, gọi c(P) = $x_3x_2x_1x_0$, với x_i bằng 0 hoặc 1, là mã này.

0110	0010	0011
0100	0000	0001
1100	1000	1001

Hình 4.2. Mã hóa các khu vực trong thuật toán cắt đường thẳng Cohen-Sutherland.

Các bit x_i có ý nghĩa như sau:

x_0 = 1 khi và chỉ khi P nằm hoàn toàn bên phải của đường biên phải.

x_1 = 1 khi và chỉ khi P nằm hoàn toàn bên trên của đường biên trên.

x_2 = 1 khi và chỉ khi P nằm hoàn toàn bên trái của đường biên trái.

x_3 = 1 khi và chỉ khi P nằm hoàn toàn bên dưới của đường biên dưới.

Thuật toán có ba bước:

Bước 1. Mã hóa P1 và P2. Đặt c1 = c(P1) và c2 = c(P2).

Bước 2. Kiểm tra nếu một đoạn có thể bị loại bỏ một cách dễ dàng sử dụng toán tử AND và OR bit để kiểm tra xem

(a) c1 OR c2 = 0, hoặc

(b) c1 AND c2 ≠ 0.

Trong trường hợp (a), đoạn đó nằm hoàn toàn trong cửa sổ vì cả hai đầu của đoạn nằm trong cửa sổ và cửa sổ là hình lồi. Trả về Q1 = P1 và Q2 = P2.

Trong trường hợp (b), đoạn đó nằm hoàn toàn ngoài cửa sổ. Đây là vì cả hai đầu của đoạn nằm ở một nửa mặt phẳng không chứa cửa sổ. Trả về đoạn rỗng.

Bước 3. Nếu một đoạn không bị loại bỏ một cách đơn giản, chúng ta sẽ chia đoạn đó ra. Sau đó chúng ta lại quay lại Bước 1 với những đoạn mới với quy trình như sau:

(a) Trước hết, tìm điểm đầu mút P sẽ xác định đường sẽ sử dụng để cắt.

Nếu c1 = 0000, thì P1 không cần phải clip và đặt P là P2 và Q là P1.

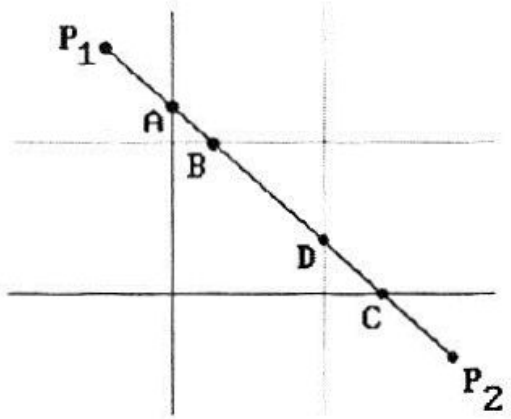
Nếu c1 ≠ 0000, đặt P là P1 và Q là P2.

(b) Đường thẳng dùng để clip được xác định bằng bit bên trái nhất mà khác 0 trong c(P). Ví dụ

```
trong Hình 4.3, P = P1, Q = P2, và đường thẳng
dùng để clip là biên trái của cửa sổ. Đặt A là
giao của đoạn [P,Q] với đường này.

(c) Lặp lại Bước 1-3 cho đoạn [A,Q].
```

Thuật toán sẽ kết thúc ở Bước 2.

Với thuật toán Cohen-Sutherland, lưu ý rằng việc mã hóa được thực hiện rất dễ dàng thông qua việc so sánh tọa độ của P1 và P2 với các hằng số với giả thiết rằng các đường biên của cửa sổ song song với trục tung và trục hoành. Chỉ ở Bước 3 mới thực sự cần các tính toán. Trong trường hợp xấu nhất, chúng ta phải tiến hành cắt 4 lần (xem Hình 4.3).



Hình 4.3. Một ví dụ về thuật toán cắt xén đường thẳng Cohen-Sutherland.

Dưới đây là phần mã mô tả thuật toán Cohen-Sutherland

```
/*Hằng số */
word BIEN_PHAI = 1;
word BIEN_TREN = 2;
word BIEN_TRAI = 4;
word BIEN_DUOI = 8;
/*
Hàm này cắt đoạn từ (x0, y0) đến (x1, y1) với của
số [xmin, xmax], [ymin, ymax]. Nó trả về false nếu
đoạn này nằm hoàn toàn ngoài của sổ và true nếu
ngược lại. Trong trường hợp hàm trả về true, các
```

```
biến x0, y0, x1, và y1 sẽ được thay bằng tọa độ
của đoạn sau khi bị cắt.
*/
bool CS_Clip (float &x0, float &y0, float &x1,
float &y1, float xmin, float ymin, float xmax,
float ymax)
{
      word c0, c1, c;
      float x, y;

      /* trước hết, mã hóa hai điểm */
      c0 = RegionCode (x0,y0);
      c1 = RegionCode (x1,y1);

      /* Vòng lặp chính */
      while (c0 | c1 != 0)
      {

            if (c0 & c1 != 0) {
                  return false;
            }
            else {
                  /*Chọn điểm đầu tiên không ở
            trong cửa sổ*/
                  c = c0;
                  if (c == 0) c = c1;

                  /*Cắt với đường thẳng tương ứng
                  với bit khác không đầu tiên */
                  if (BIEN_TRAI & c !=0) {
                        //  Cắt với biên trái
                        x = xmin;
                        y = y0 + (y1 - y0)*(xmin -
                        x0)/(x1 - x0);
                  }
                  else if (BIEN_PHAI & c != 0) {
                        /* Cắt với biên phải */
                        x = xmax;
                        y := y0 + (y1 -y0)*(xmax -
                  x0)/(x1 - x0);
```

```
                    }
                    else if (BIEN_DUOI & c !=0) {
                          /* Cắt với biên dưới */
                          x := x0 + (x1 - x0)*(ymin -
                    y0)/(y1 - y0);
                          y := ymin;
                    }
                    else if (BIEN_TREN & c !=0) {
                          /* Cắt với biên trên */
                          x := x0 + (x1 - x0)*(ymax -
                    y0)/(y1 - y0);
                          y := ymax;
                    }

          /*Cập nhật đầu của đoạn được cắt và mã
của nó */
                    if (c == c0) {
                          x0 = x;
                          y0 = y;
                          c0 = RegionCode (x0,y0);
                    }
                    else {
                          x1 = x;
                          y1 = y;
                          c1 = RegionCode (x1,y1);
                    }
               }
      } // while
      return true;
}

/* Trả về mã hóa 4-bit của điểm (x,y) */
word RegionCode (float x, floaty)
{
      word c;
      c = 0;
      if (x < xmin) c = c + BIEN_TRAI;
      else if (x > xmax) c = c + BIEN_PHAI;
```

```
        if (y < ymin) c = c + BIEN_DUOI;
        else if (y > ymax) c = c + BIEN_TREN;
        return c;
}
```

4.1.2. Thuật toán Cắt xén đoạn thẳng Cyrus-Beck

Thuật toán cắt xén đoạn thẳng Cyrus-Beck [CyrB78] cắt một đoạn S với một đa giác lồi X bất kỳ. Đặt $S = [P_1,P_2]$ và $X = Q_1Q_2 \ldots Q_k$. Vì giả thiết X là lồi, nó là giao điểm của các nửa mặt phẳng xác định bởi các cạnh của nó. Chính xác hơn, với mỗi cạnh $[Q_i,Q_{i+1}]$, $i = 1,2, \ldots, k$, (Q_{k+1} là điểm Q_1), chúng ta có thể chọn một véc-tơ pháp tuyến, sao cho X có thể được mô tả ở dạng

$$X = \bigcap_{i=1}^{k} H_i \tag{4.1}$$

Với H_i là nửa mặt phẳng:

$$H_i = \{Q \mid N_i \bullet (Q - Q_i) \geq 0\}. \tag{4.2}$$

Với lựa chọn này, véc-tơ pháp tuyến sẽ trỏ vào đa giác.

Từ công thức trên, ta có:

$$S \cap X = \bigcap_{i=1}^{k} (S \cap H_i). \tag{4.3}$$

Nói một cách khác, ta có thể cắt đoạn S bằng X bằng cách cắt nó với từng nửa mặt phẳng H_i. Đây chính là ý tưởng cơ bản thứ nhất đằng sau thuật toán Cyrus-Beck. Ý tưởng thứ hai là việc sử dụng phương trình tham số $P_1 + tP_1P_2$ để biểu diễn đường thẳng L chứa đoạn S và dùng nó để cắt với giá trị t.

Gọi L_i là đường thẳng chứa đoạn $[Q_i,Q_{i+1}]$. Khoảng $I_i = [a_i, b_i]$ được xác định như sau:

Trường hợp 1: L song song với L_i.

(a) Nếu L nằm hoàn toàn trong H_i, đặt $I_i = (-\infty, +\infty)$.

(b) Nếu L nằm hoàn toàn ngoài H_i, đặt I_i = rỗng.

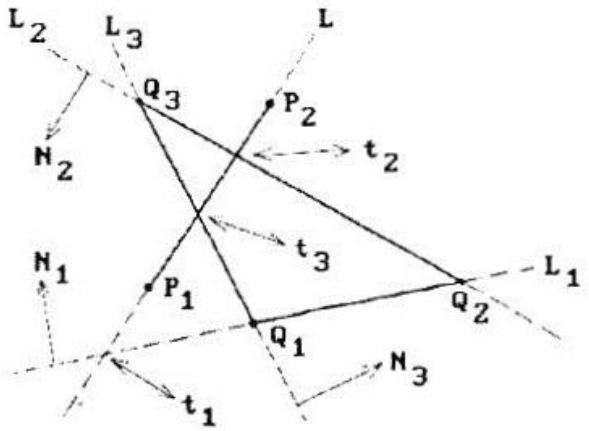
Trường hợp 2: L không song song với L_i.

Trong trường hợp này L sẽ cắt Li tại điểm $P = P_1 + t_i P_1 P_2$. Chúng ta phân biệt trường hợp khi L "đi vào" mặt phẳng H_i và khi L "đi ra" khỏi H_i

(a) (Đi vào) Nếu $P_1P_2 \bullet N_i \geq 0$, thì đặt $I_i = [t_i, +\infty)$.

(b) (Đi ra) Nếu $P_1P_2 \bullet N_i < 0$, thì đặt $I_i = (-\infty, t_i]$.

Hình 4.4 mô tả đoạn $S = [P_1, P_2]$ được cắt bởi tam giác $Q_1Q_2Q_3$.



Hình 4.4. Minh họa thuật toán cắt xén Cyrus-Beck.

Lưu ý là việc tìm giao điểm P trong trường hợp 2 là tương đối dễ dàng. Mọi việc phải làm là giải phương trình

$$N_i \bullet (P_1 + tP_1P_2 - Q_i) = 0 \qquad \textbf{(4.4)}$$

cho t.

Trước hết, đặt $I_0 = [a_0,b_0] = [0,1]$. Khoảng I_0 là tập hợp các tham số của các điểm trên đường thẳng L nằm trên S. Dễ dàng nhận thấy đoạn

$$\begin{aligned} I &= \bigcap_{i=0}^{k} I_i \\ &= [\max_{0\le i\le k} a_i, \min_{0\le i\le k} b_i] \\ &= [a, b] \end{aligned} \tag{4.5}$$

là tập hợp tham số của các điểm trong đoạn $S \cap X$. Nói một cách khác, nếu I không rỗng, thì:

$$S \cap X = [P_1 + aP_1P_2, P_1 + bP_1P_2] \tag{4.6}$$

Chúng ta hãy thử xem ví dụ trong Hình 4.4. Trong ví dụ này,

$$I = [0,1] \cap [t_1,+\infty) \cap (-\infty,t_2] \cap [t_3,+\infty) = [t_3,t_2], \tag{4.7}$$

rõ ràng là một câu trả lời đúng.

4.1.3. Thuật toán Cắt xén đoạn thẳng Liang-Barsky

Thuật toán cắt xén đoạn thẳng Liang-Barsky [LiaB83] tối ưu hóa thuật toán cắt xén đoạn thẳng Cyrus-Beck trong trường hợp khung cắt là hình chữ nhật. Thuật toán này bắt đầu bằng việc xét phương trình tham số của đường thẳng chứa đoạn P_1P_2 với $P_1 = (x_1,y_1)$ và $P_2 = (x_2,y_2)$:

$$P = P_1 + tP_2 \tag{4.8}$$

Hay:

$$\begin{aligned} x &= x_1 + t^*\Delta x \\ y &= y_1 + t^*\Delta y \end{aligned} \tag{4.9}$$

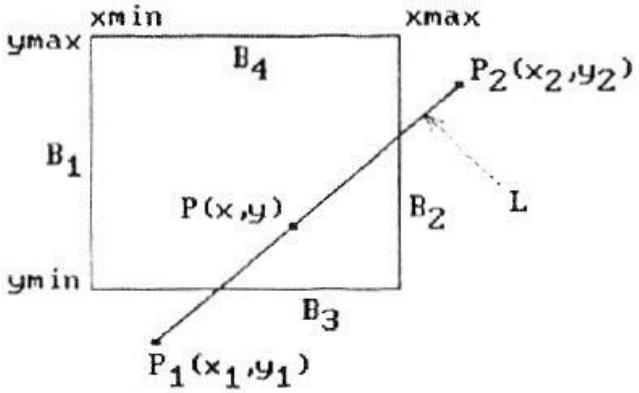
với $\Delta x = x_2 - x_1$ và $\Delta y = y_2 - y_1$.

Nếu cửa sổ cắt W là hình chữ nhật $[x_{min},x_{max}]\times[y_{min},y_{max}]$ thì điểm P thuộc về W khi và chỉ khi:

$$x_{min} \le x_1 + \Delta x t \le x_{max}$$
$$y_{min} \le y_1 + \Delta y t \le y_{max} \qquad \textbf{(4.10)}$$

Có nghĩa là:

$$-\Delta x t \le x_1 - x_{min}$$
$$\Delta x t \le x_{max} - x_1$$
$$-\Delta y t \le y_1 - y_{min}$$
$$\Delta y t \le y_{max} - y_1 \qquad \textbf{(4.11)}$$



Hình 4.5. Minh họa thuật toán cắt xén Liang-Barsky.

Để đơn giản, chúng ta sẽ giới thiệu các biến phụ c_k và q_k, và viết lại những phương trình trên như sau:

$$c_1 t \le q_1$$
$$c_2 t \le q_2$$
$$c_3 t \le q_3$$
$$c_4 t \le q_4 \qquad \textbf{(4.12)}$$

Đặt $t_k = q_k/c_k$ với $c_k \neq 0$. Đặt B_1, B_2, B_3, và B_4 là các biên trái, phải, dưới và trên của cửa sổ cắt. Chúng ta có thể thấy:

(1) Nếu $c_k > 0$, đường thẳng L đi từ phía trong ra phía ngoài của đường biên B_k khi t tăng, và chúng ta gọi t_k là điểm ra.

(2) Nếu $c_k < 0$, thì đường thẳng L đi từ phía ngoài vào phía trong của đường biên B_k khi t tăng và ta gọi t_k là điểm vào.
(3) Nếu $c_k = 0$, đường thẳng L song song với B_k, và ngoài cửa số nếu $q_k < 0$.

Sau đây là thuật toán:

```
Loại bỏ đoạn thẳng nếu
     Một giá trị vào (t ứng với điểm vào) lớn hơn 1
hoặc
     giá trị ra (t ứng với điểm ra) nhỏ hơn 0 hoặc
     một giá trị vào lớn hơn giá trị ra
Nếu không, đoạn thẳng sẽ giao với cửa sổ cắt. Chúng
ta cần tính đoạn giao chỉ khi t0 > 0 và t1 < 1, khi
     t0 = max (0, max { các giá trị vào tk } ), và
     t1 = min (1, min { các giá trị ra tk } ).

(Trường hợp t0 = 0 và t1 = 1 có nghĩa là ta sẽ dùng
đầu mút của đoạn, không cần cắt).
```

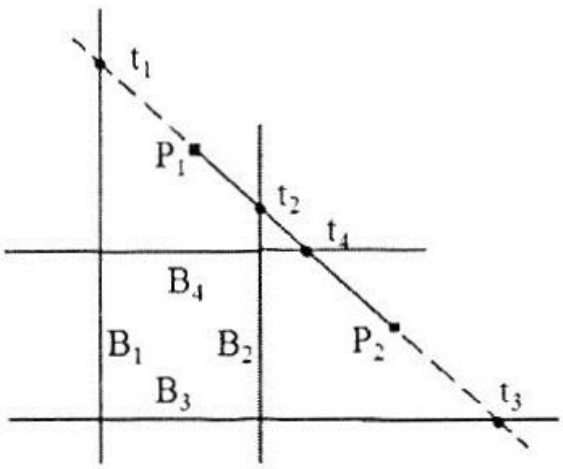
Sau đây là cài đặt của thuật toán:

```
/* Hàm này cắt đoạn từ (x0,y0) đến (x1,y1) bằng cửa
sổ [xmin, xmax]x[ymin, ymax]. Nó trả về false nếu
đoạn đó nằm hoàn toàn  ngoài cửa sổ, và trả về true
trong trường hợp ngược lại. Trong trường hợp trả về
true, các biến x0, y0, x1, và y1 sẽ được thay đổi
để mô tả đoạn sau khi bị cắt.*/


bool  LB_Clip(float  &x0,float  &y0,float  &x1,float
&y1,float xmin, float ymin,float xmax,float ymax)
{
     float t0,t1,dx,dy;
     bool more;
     t0 = 0; t1 = 1; dx = x1 - x0;
     Findt(-dx,x0 - xmin,t0,t1,more); // biên trái
```

```
    if (more)
    {
        Findt(dx,xmax - x0,t0,t1,more); // biên
    phải
        if (more)
        {
            dy = y1 - y0;
            Findt(-dy,y0-ymin,t0,t1,more);//
        biên dưới
            if (more)
            {
                Findt(dy,ymax-
            y0,t0,t1,more);//biên trên
                if (more)
                { // cắt đoạn thẳng
                    if (t1 < 1)
                    { // tính điểm thoát ra
                        x1= x0 + t1*dx;
                        y1= y0 + t1*dy;
                    }
                    if (t0 > 0)
                    { // tính điểm vào
                        x0= x0 + t0*dx;
                        y0= y0 + t0*dy;
                    }
                }
            }
        }
    }
    return(more);
    }

void Findt(float denom,float num,float &t0,float
&t1,bool &more)
```

```
{
      float r;
      more = true;
      if (denom < 0)
      { // đoạn thẳng từ ngoài vào trong
            r = num/denom;
            if (r > t1)
                  more = false;
            else if (r > t0) t0 = r;
            }
      else if (denom > 0)
      { // đoạn thẳng từ trong ra ngoài
            r = num/denom;
            if (r < t0)
                  more = false;
            else if (r < t1) t1 = r;
      } else if (num < 0)
      { // đoạn thẳng song song với đường biên
            more = false;
            }
} // Findt
```

Hình 4.6 cho thấy một ví dụ về thuật toán cắt xén Liang-Barsky với các giá trị *t* tìm được.



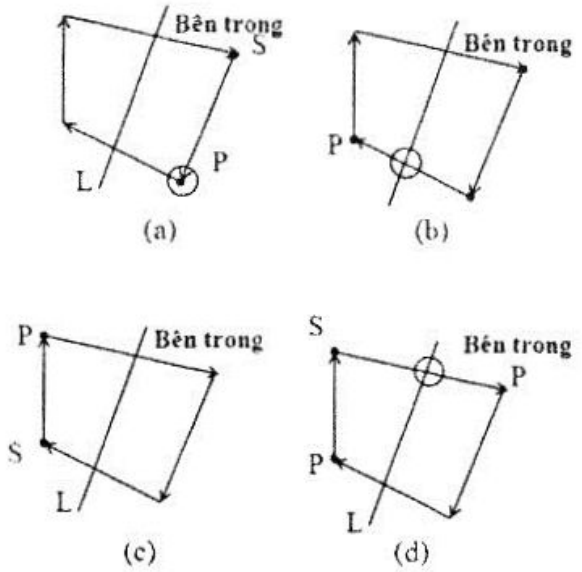
Hình 4.6. Ví dụ về thuật toán cắt xén Liang-Barsky với các giá trị *t* tìm được.

4.2. Các thuật toán Cắt xén đa giác

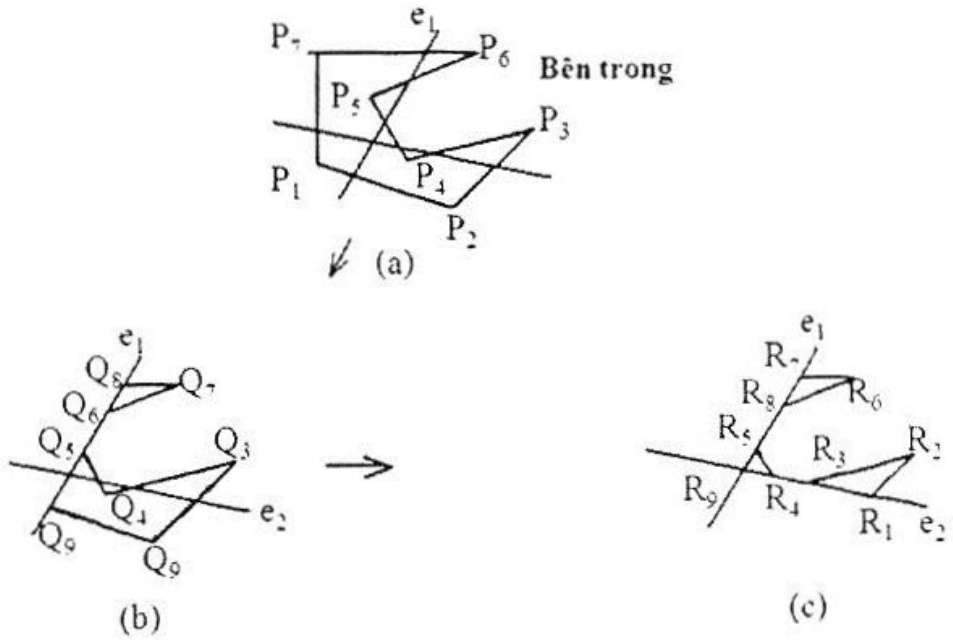
4.2.1. Thuật toán Cắt xén đa giác Sutherland-Hodgman

Một trong những thuật toán cắt xén đa giác đầu tiên là thuật toán Sutherland-Hodgman [SutH74]. Nó được thực hiện thông qua việc cắt toàn bộ đa giác với một cạnh của cửa sổ (chính xác hơn là với nửa mặt phẳng chứa đa giác cắt và xác định bởi cạnh đó), sau đó cắt đa giác mới với cạnh tiếp theo, và tiếp tục như vậy cho đến khi đa giác được cắt bằng cả bốn cạnh của cửa sổ. Một ưu điểm quan trọng của thuật toán này là tránh được việc sinh ra những dữ liệu trung gian.

Giả thiết rằng chúng ta muốn cắt một đa giác bằng một cạnh **e**, với đa giác được thể hiện qua một chuỗi các đỉnh $\mathbf{P}_1, \mathbf{P}_2, \ldots, \mathbf{P}_n$. Thuật toán mỗi lần xét một đỉnh đầu vào $\mathbf{P}_i$ và sinh ra một chuỗi các đỉnh mới $\mathbf{Q}_1, \mathbf{Q}_2, \ldots, \mathbf{Q}_m$. Mỗi $\mathbf{P}_i$ sinh ra 0, 1, hay 2 đỉnh $\mathbf{Q}_j$, tùy thuộc vào vị trí của các đỉnh đầu vào đối với **e**. Nếu ta đang coi một đỉnh đầu vào **P** là đỉnh kết thúc của một cạnh nối với đỉnh **S** trước đó thì các đỉnh **Q** được sinh ra phụ thuộc vào mối quan hệ giữa cạnh [**S**,**P**] với đường thẳng **L** đi qua cạnh **e**. Có bốn trường hợp xảy ra (Xem Hình 4.7). Phần nửa mặt phẳng chứa cửa sổ được đánh dấu là "bên trong". Những đỉnh được khoanh tròn là những đỉnh đầu ra. Hình 4.8 cho thấy một ví dụ về sự hoạt động của thuật toán. Cắt đa giác với các đỉnh $\mathbf{P}_i$ bằng cạnh $\mathbf{e}_1$, ta thu được đa giác với các đỉnh $\mathbf{Q}_i$. Cắt đa giác với các đỉnh $\mathbf{Q}_i$ bằng cạnh $\mathbf{e}_2$, ta thu được đa giác $\mathbf{R}_i$.



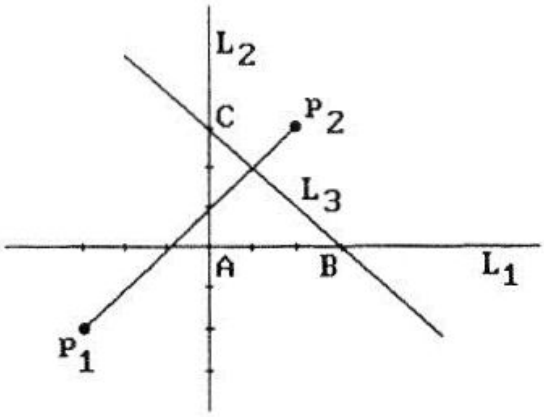
Hình 4.7. Bốn trường hợp trong thuật toán Sutherland-Hodgman.



Hình 4.8. Minh họa thuật toán cắt xén đa giác Sutherland-Hodgman.

Câu hỏi và bài tập

1. Hãy nêu ưu điểm của thuật toán cắt xén đoạn thẳng Cohen-Sutherland.
2. Việc kiểm tra mã của 2 đầu đoạn thẳng trong thuật toán hóa Cohen-Sutherland còn chưa giải quyết được trường hợp nào?
3. Hãy nêu ưu điểm của thuật toán cắt xén đa giác Sutherland-Hodgman.
4. Hãy tổng quát hóa thuật toán cắt xén đa giác Sutherland-Hodgman để xử lý các trường hợp đặc biệt là một đỉnh của đa giác nằm trên cạnh của cửa sổ cắt.
5. Cho $p_1 = (-4, -2)$ và $p_2 = (2, 3)$. Cho A = (0, 0), B = (3, 0) và C = (0, 3). Trình bày các bước của thuật toán Cyrus-Beck và tính $[a_i, b_i]$ sinh ra khi cắt đoạn $[p_1, p_2]$ bằng tam giác ABC. Xem hình dưới đây.



Giả sử các đường thẳng L_1, L_2 và L_3 được xác định bởi các phương trình tương ứng là $y = 0$, $x = 0$ và $x + y = 3$.

Bài tập lập trình: Chương trình vẽ - hãy mở rộng chương trình đã phát triển trong bài tập lập trình ở chương trước để cho phép người sử dụng vẽ bằng các đoạn thẳng trong một khung ảnh hình chữ nhật và tô màu các vùng khép kín. Trước hết, xác định một khung ảnh hình chữ nhật. Sau đó bắt các sự kiện chuột để xác định các đoạn thẳng người sử dụng muốn vẽ. Nếu có phần nào của đoạn thẳng nằm ngoài khung ảnh, hãy cài đặt một thuật toán cắt xén để loại bỏ phần nằm ngoài đó.

Chương 5
CÁC PHÉP CHIẾU VÀ PHÉP BIẾN ĐỔI

Trong chương này, chúng ta sẽ làm quen với các phép biến đổi các vật thể trong không gian ba chiều và các phép chiếu. Ở đây, vì khái niệm các hệ tọa độ rất quan trọng, trước hết chúng ta sẽ phân biệt rõ một số hệ tọa độ. Hình 5.1 cho thấy quan hệ giữa các hệ tọa độ trong luồng xử lý đồ họa từ giai đoạn mô tả đối tượng đến khi hiển thị được lên màn hình.

Hệ tọa độ thế giới (***The World Coordinate System***). Đây là hệ tọa độ mà người sử dụng dùng để định nghĩa các đối tượng.

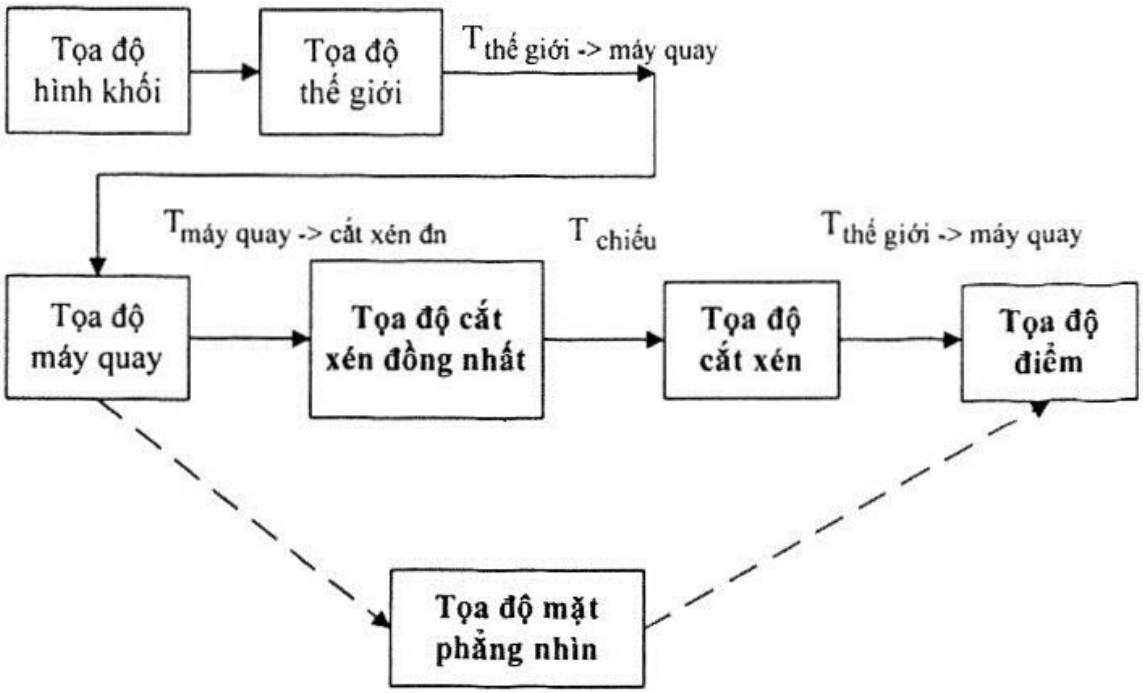
Hệ tọa độ hình dạng (***The Shape Coordinate System***). Đây là hệ tọa độ dùng để định nghĩa hình dạng. Hệ tọa độ này có thể rất khác với hệ tọa độ thế giới. Việc sử dụng hệ tọa độ này tạo thuận lợi rất nhiều cho việc mô tả các đối tượng. Ví dụ, một hình e-lip có tâm ở gốc tọa độ có phương trình khá đơn giản, trong khi đó một hình e-lip có tâm ở vị trí khác có phương trình phức tạp hơn nhiều. Ngoài ra, nếu muốn hình e-lip đó quay đi một góc 45 độ thì phương trình còn phức tạp hơn nữa. Hệ tọa độ hình dạng chuyên dùng để xác định các hình dạng, sau đó kết hợp các phép biến đổi như tịnh tiến và quay trong hệ tọa độ thế giới giúp cho việc định nghĩa các đối tượng đơn giản hơn rất nhiều.

Hệ tọa độ máy quay (***The Camera Coordinate System***). Toàn cảnh thế giới thu được thông qua phép chiếu lên một mặt phẳng được gọi là **cái nhìn phối cảnh** (*perspective view*). Để mô tả về phép chiếu này, chúng ta sẽ sử dụng một số ý tưởng của máy quay. Khi chụp một bức ảnh, máy quay được đặt tại một vị trí nhất định

và hướng về một hướng nào đó. Một hướng lên trên của máy quay xác định hướng lên trên của ảnh. Chúng ta coi mặt phẳng nhìn như là phim của máy quay với một sự khác biệt là phim của máy quay ở sau máy quay và ảnh trên phim ngược chiều với cảnh còn mặt phẳng nhìn ở trước máy quay và ảnh trên mặt phẳng nhìn cùng chiều với cảnh. Như vậy, với một máy quay trong đồ họa máy tính, chúng ta có những tham số sau:

- một vị trí **p**
- một hướng nhìn **v** (hướng của máy quay)
- một hướng lên **w**
- một khoảng cảnh **d** từ máy quay đến mặt phẳng nhìn

Bây giờ, phép chiếu phối cảnh từ máy quay đã được định nghĩa rõ ràng. Chúng ta có thể nhìn thế giới từ bất cứ điểm **p** nào, theo bất cứ hướng **v** nào, và hướng nào là phía trên của bức ảnh. Tham số **d** giúp chúng ta phóng to/thu nhỏ cảnh vật một cách dễ dàng.



Hình 5.1. Luồng chuyển qua các hệ tọa độ.

5.1. Các phép biến đổi

Có bốn phương pháp cơ bản để thay đổi các đối tượng: thay đổi vị trí, thay đổi hướng, thay đổi kích thước và bóp méo. Những sự thay đổi này được gọi là các phép biến đổi:

- Tịnh tiến
- Quay
- Co giãn
- Kéo

Chúng ta có thể tịnh tiến một điểm trong mặt phẳng (x,y) tới vị trí mới bằng cách cộng thêm lượng cần tịnh tiến vào các tọa độ của điểm đó. Với mỗi điểm P(x,y) cần chuyển đi d_x đơn vị theo trục x và d_y đơn vị theo trục y đến điểm P'(x',y'), ta có:

$$x' = x + d_x,\ y' = y + d_y \qquad \textbf{(5.1)}$$

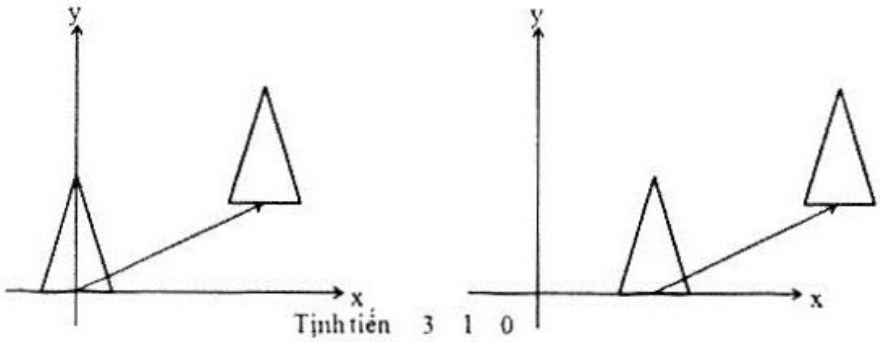
Nếu ta định nghĩa các véc-tơ cột

$$P = \begin{bmatrix} x \\ y \end{bmatrix}, \qquad P' = \begin{bmatrix} x' \\ y' \end{bmatrix}, \qquad T = \begin{bmatrix} d_x \\ d_y \end{bmatrix} \qquad \textbf{(5.2)}$$

thì Công thức 5.1 có thể được trình bày lại dưới dạng

$$P' = P + T \qquad \textbf{(5.3)}$$

Chúng ta có thể tịnh tiến cả vật thể bằng cách áp dụng Công thức **5.1** với mọi điểm của vật thể. Tuy nhiên, mỗi đối tượng có vô số điểm. May mắn là chúng ta có thể tịnh tiến các đoạn thẳng, thậm chí cả đa giác, chỉ bằng cách tính kết quả của phép tịnh tiến với các đỉnh của chúng. Điều này cũng đúng với phép quay và phép co giãn. Hình 5.2 cho thấy tác động của phép tịnh tiến với một vật thể.



Hình 5.2. Tác động của phép tịnh tiến với một vật thể.

Một điểm P(x,y) có thể được co giãn một lượng s_x theo trục x và một lượng s_y theo trục y thành điểm P'(x',y') bằng phép nhân

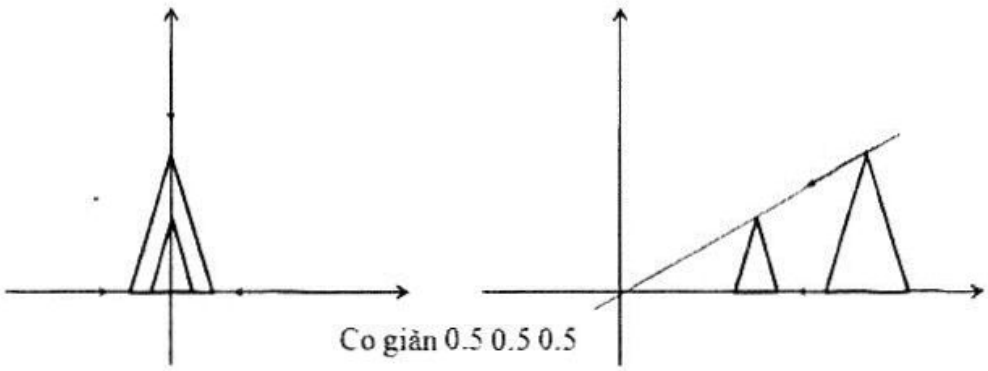
$$x' = s_x . x,\ y' = s_y . y \qquad \textbf{(5.4)}$$

Ở dạng ma trận

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix} . \begin{bmatrix} x \\ y \end{bmatrix} \qquad hay \qquad P' = S.P \qquad \textbf{(5.5)}$$

với S là ma trận vuông trong phần đầu của Công thức 5.5.

Trong Hình 5.3, vật thể được co giãn với tỉ lệ 0.5 theo cả x và y. Lưu ý rằng, phép co giãn này thực hiện quanh gốc tọa độ. Kết quả là, vật thể bé hơn và gần gốc tọa độ hơn. Nếu các hệ số co giãn lớn hơn 1 thì vật thể sẽ to hơn và xa gốc tọa độ hơn. Lưu ý rằng tỷ lệ các chiều của vật thể cũng bị thay đổi do s_x khác s_y. Tỷ lệ này sẽ được giữ nguyên nếu $s_x = s_y$.



Hình 5.3. Một tam giác được co giãn với tỉ lệ 0.5 theo cả x và y.

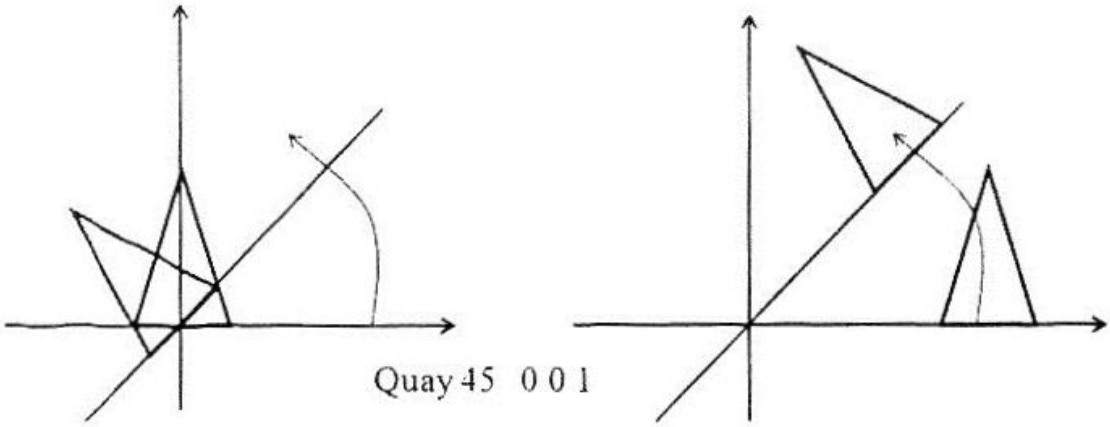
Một điểm P(x,y) có thể được quay quanh gốc tọa độ một góc θ, được xác định thông qua

$$x' = x.\cos\theta - y.\sin\theta, \qquad y' = x.\sin\theta + y.\cos\theta \qquad \textbf{(5.6)}$$

Ở dạng ma trận, chúng ta có

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} . \begin{bmatrix} x \\ y \end{bmatrix} \qquad hay \qquad P' = R.P \qquad \textbf{(5.7)}$$

Với ***R*** là ma trận quay trong Công thức 5.7. Hình 5.4 cho thấy phép quay một vật thể đi 45 độ.



Hình 5.4. Phép quay một tam giác đi 45 độ.

Góc quay dương được tính theo chiều ngược kim đồng hồ từ **x** đến **y**.

5.1.1. Tọa độ đồng nhất và biểu diễn ma trận của các phép biến đổi 2D

Cách thể hiện ma trận của phép tịnh tiến (phép cộng) không giống như phép quay và phép co giãn (phép nhân). Chúng ta muốn xử lý ba phép biến đổi này cùng bằng phép nhân để có thể kết hợp chúng lại dễ dàng. Nếu các điểm được biểu diễn dưới dạng **tọa độ đồng nhất** (*homogeneous coordinate*), tất cả các phép biến đổi có thể biểu diễn dưới dạng phép nhân.

Trong tọa độ đồng nhất, ta thêm một tọa độ thứ ba cho mỗi điểm. Thay vì được thể hiện bằng hai tọa độ (x,y), mỗi điểm sẽ được thể hiện bằng bộ ba (x,y,W). Chúng ta sẽ coi hai bộ ba (x,y,W) và (x',y',W') thể hiện cùng một điểm nếu chúng là bội số của nhau. Như vậy, (2,3,6) và (4,6,12) là hai bộ ba biểu diễn cùng một điểm. Mỗi điểm sẽ có vô số biểu diễn tọa độ đồng nhất khác nhau với điều kiện phải có ít nhất một tọa độ khác không: (0,0,0) không phải là một tọa độ hợp lệ. Nếu tọa độ W khác không, chúng ta có thể chia toàn bộ các tọa độ cho W: (x,y,W) và (x/W, y/W,1) biểu diễn cùng một điểm (x/W,y/W) trên hệ tọa độ Đề-các. Các điểm với W = 0 là các điểm ở vô cùng.

Như vậy, các điểm bây giờ được biểu diễn dưới dạng véc-tơ cột có ba phần tử. Do đó, ma trận biến đổi để nhân với một véc-tơ

điểm để ra một véc-tơ điểm khác, phải có kích thước 3x3. Trong hệ tọa độ đồng nhất, phép tịnh tiến trong Công thức **5.1** được thể hiện bởi:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & d_x \\ 0 & 1 & d_y \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \quad \textbf{(5.8)}$$

Công thức 5.8 có thể viết lại dưới dạng:

$$P' = T(d_x, d_y)P \quad \textbf{(5.9)}$$

trong đó

$$T(d_x, d_y) = \begin{bmatrix} 1 & 0 & d_x \\ 0 & 1 & d_y \\ 0 & 0 & 1 \end{bmatrix} \quad \textbf{(5.10)}$$

Điều gì xảy ra nếu một điểm P được tịnh tiến bởi $T(d_{x1}, d_{y1})$ đến P' và sau đó tịnh tiến bởi $T(d_{x2}, d_{y2})$ đến P''? Kết quả sẽ là phép tịnh tiến $T(d_{x1} + d_{x2}, d_{y1} + d_{y2})$. Chúng ta sẽ kiểm tra với công thức ma trận trên:

$$P' = T(d_{x1}, d_{y1})P \quad \textbf{(5.11)}$$

$$P'' = T(d_{x2}, d_{y2})P' \quad \textbf{(5.12)}$$

Thay Công thức 5.11 vào Công thức 5.12, chúng ta có

$$P'' = T(d_{x2}, d_{y2})(T(d_{x1}, d_{y1})P) = (T(d_{x2}, d_{y2})T(d_{x1}, d_{y1}))P \quad \textbf{(5.13)}$$

Ma trận tích $T(d_{x2}, d_{y2}) . T(d_{x1}, d_{y1})$ là

$$\begin{bmatrix} 1 & 0 & d_{x1} \\ 0 & 1 & d_{y1} \\ 0 & 0 & 1 \end{bmatrix}\begin{bmatrix} 1 & 0 & d_{x2} \\ 0 & 1 & d_{y2} \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & d_{x1}+d_{x2} \\ 0 & 1 & d_{y1}+d_{y2} \\ 0 & 0 & 1 \end{bmatrix} \quad \textbf{(5.14)}$$

và đúng là kết quả của hai phép tịnh tiến là phép tịnh tiến $T\left(d_{x1}+d_{x2}, d_{y1}+d_{y2}\right)$. Ma trận tích thường được gọi là ma trận kết hợp.

Tương tự, phép co giãn trong Công thức 5.5 được thể hiện ở dạng ma trận như sau:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix}\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \quad \textbf{(5.15)}$$

Định nghĩa

$$S\left(s_x, s_y\right) = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad \textbf{(5.16)}$$

Chúng ta có

$$P' = S\left(s_x, s_y\right)P \quad \textbf{(5.17)}$$

Chúng ta cũng mong muốn hai phép co giãn liên tiếp sẽ là một phép nhân. Cho

$$P' = S\left(s_{x1}, s_{y1}\right)P \quad \textbf{(5.18)}$$

$$P'' = S\left(s_{x2}, s_{y2}\right)P' \quad \textbf{(5.19)}$$

Thay Công thức **5.18** vào Công thức **5.19** ta có

$$P'' = S(s_{x2}, s_{y2}).(S(s_{x1}, s_{y1}).P) = (S(s_{x2}, s_{y2}).S(s_{x1}, s_{y1})).P \quad \textbf{(5.20)}$$

Ma trận tích $S(s_{x2}, s_{y2}).S(s_{x1}, s_{y1})$ là

$$\begin{bmatrix} s_{x1} & 0 & 0 \\ 0 & s_{y1} & 0 \\ 0 & 0 & 1 \end{bmatrix}.\begin{bmatrix} s_{x2} & 0 & 0 \\ 0 & s_{y2} & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} s_{x1}.s_{x2} & 0 & 0 \\ 0 & s_{y1}.s_{y2} & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad \textbf{(5.21)}$$

Như vậy, phép co giãn đúng là theo phép nhân.

Cuối cùng, phép quay trong Công thức 5.7 có thể biểu diễn bằng

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}.\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \quad \textbf{(5.22)}$$

Đặt

$$R(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad \textbf{(5.23)}$$

chúng ta có

$$P' = R(\theta).P \quad \textbf{(5.24)}$$

Tương tự chúng ta cũng có thể chứng minh được kết quả hai phép quay liên tiếp là một phép quay của tổng góc.

Trong ma trận con 2x2 của ma trận trong Công thức 5.23, con hai dòng là hai véc-tơ. Các véc-tơ này có những tính chất sau:

a. Mỗi véc-tơ là một véc-tơ đơn vị
b. Các véc-tơ này vuông góc với nhau (tích vô hướng bằng không)

c. Véc-tơ thứ nhất và thứ hai được quay đi bởi phép quay R(θ) để nằm trên trục x dương và trục y dương.

Hai tính chất đầu tiên cũng đúng với các cột của ma trận con 2x2 này. Một ma trận có những tính chất này được gọi là **trực giao đặc biệt** (*special orthogonal*).

Một ma trận biến đổi có dạng

$$\begin{bmatrix} r_{11} & r_{12} & t_x \\ r_{21} & r_{22} & t_y \\ 0 & 0 & 1 \end{bmatrix} \qquad (5.25)$$

với ma trận con trái trên 2x2 là trực giao, thì bảo toàn được góc và độ dài. Các phép biến đổi như vậy được gọi là các phép biến đổi **hình thể cứng** (*rigid-body*) vì hình thể của các vật không bị bóp méo sau phép biến đổi. Tích của một chuỗi bất kỳ các ma trận của phép quay và phép tịnh tiến tạo ra một ma trận dạng này.

Phép quay, phép tịnh tiến và phép co giãn được gọi là các biến đổi afin, với tính chất bảo toàn các đường song song, nhưng chưa chắc bảo toàn độ dài và góc. Một phép biến đổi cơ bản khác là **phép kéo** (*shear*), cũng là một phép biến đổi afin. Có hai phép kéo: kéo theo trục x và kéo theo trục y. Hình 5.5 cho thấy hiệu ứng của phép kéo theo mỗi trục.

Hình 5.5. Một vật được kéo theo trục x và theo trục y.

Ma trận biến đổi của phép kéo theo trục **x** là:

$$SH_x = \begin{bmatrix} 1 & a & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad (5.26)$$

với **a** là hằng số tỉ lệ. Kết quả của phép biến đổi này với điểm (**x**,**y**) là (**x**+**ay**,**y**).

Tương tự, ma trận

$$SH_y = \begin{bmatrix} 1 & 0 & 0 \\ b & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad (5.27)$$

là ma trận biến đổi của phép kéo theo trục y.

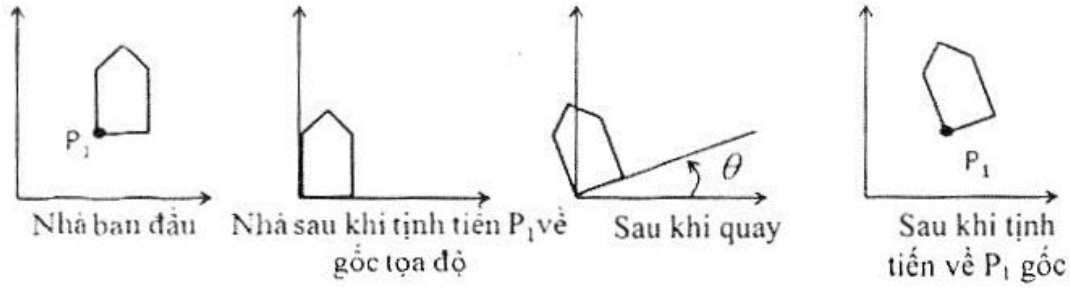
5.1.2. Kết hợp các phép biến đổi 2D

Ý tưởng của việc kết hợp được giới thiệu ở phần trước. Ở phần này chúng ta sẽ mô tả việc kết hợp các ma trận cơ bản R, S và T để tạo ra các kết quả như mong muốn. Mục đích chính của việc kết hợp các phép biến đổi là để đạt được sự hiệu quả thông qua việc áp dụng một phép biến đổi kết hợp với các điểm thay vì lần lượt áp dụng một chuỗi các phép biến đổi cho lần lượt từng điểm một.

Xét việc quay một vật thể quanh một điểm bất kỳ P_1. Vì chúng ta mới chỉ nói đến phép quay quanh gốc tọa độ, chúng ta sẽ biến bài toán khó ban đầu thành ba bài toán dễ hơn. Thật vậy, để quay quanh điểm P_1, chúng ta cần chuỗi ba phép biến đổi cơ bản:

1. Tịnh tiến sao cho P_1 trùng với gốc tọa độ
2. Quay
3. Tịnh tiến sao cho điểm tại gốc tọa độ trở về P_1.

Chuỗi phép biến đổi này được mô tả trong Hình 5.6, trong đó ngôi nhà được quay quanh điểm $P_1(x_1, y_1)$.



Hình 5.6. Quay một hình quanh một điểm bất kỳ.

Như vậy, ma trận kết hợp sẽ là $T(-x_1,-y_1).R(\theta).T(x_1,y_1)$. Sau khi tính tích của ba ma trận này, chúng ta chỉ cần áp dụng 5 lần lên 5 đỉnh của ngôi nhà thay vì áp dụng 15 phép biến đổi (mỗi đỉnh 3 phép biến đổi) nếu không dùng ma trận kết hợp.

5.1.3. Biểu diễn ma trận của các phép biến đổi 3D

Cũng như việc các phép biến đổi 2D được biểu diễn bởi các ma trận 3x3 trong tọa độ đồng nhất, các phép biến đổi 3D cũng được biểu diễn bởi các ma trận 4x4 nếu chúng ta cũng sử dụng tọa độ đồng nhất trong không gian 3 chiều. Như vậy, thay vì biểu diễn một điểm dưới dạng (x,y,z), chúng ta sẽ biểu diễn nó dưới dạng (x,y,z,W). Hai bộ bốn cùng biểu diễn một điểm nếu chúng là bội số khác không của nhau. Bộ bốn (0,0,0,0) không phải là một biểu diễn hợp lệ. Cũng như trong 2D, một biểu diễn chuẩn của (x,y,z,W) với W khác 0 là (x/W, y/W, z/W, 1). Cũng như vậy, điểm với W bằng 0 là điểm ở vô cùng.

Tịnh tiến trong 3D chỉ là mở rộng đơn giản của nó trong 2D:

$$T(d_x,d_y,d_z)=\begin{bmatrix}1 & 0 & 0 & d_x\\ 0 & 1 & 0 & d_y\\ 0 & 0 & 1 & d_z\\ 0 & 0 & 0 & 1\end{bmatrix} \qquad \textbf{(5.28)}$$

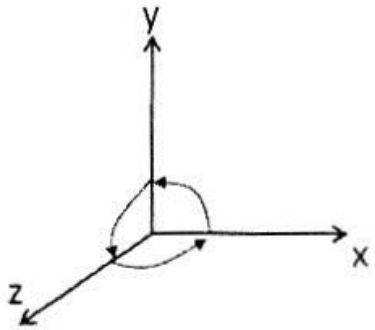
Phép co giãn cũng được mở rộng tương tự:

$$S(s_x, s_y, s_z) = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (5.29)$$

Phép quay 2D trong Công thức **5.22** chính là phép quay 3D quanh trục z với ma trận biến đổi là:

$$R_z(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (5.30)$$

với hệ tọa độ là hệ tọa độ thuận tay.



Ma trận của phép quay quanh trục **x** là:

$$R_x(\theta) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (5.31)$$

và quay quanh trục y là:

$$R_y(\theta) = \begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (5.32)$$

Các cột (và hàng) của ma trận con trái trên của $R_z(\theta)$, $R_x(\theta)$ và $R_y(\theta)$ là các véc-tơ đơn vị vuông góc từng cặp với nhau.

Các ma trận biến đổi này đều có ma trận nghịch đảo. Ma trận nghịch đảo của T có được bằng cách đổi dấu của d_x,d_y, và d_z; của S có được bằng cách thay s_x,s_y, và s_z bởi nghịch đảo của chúng; và của ba phép quay bằng cách đổi dấu của góc quay.

Tương ứng với phép kéo trong 2D, có 3 phép kéo trong 3D. Phép kéo (x,y) theo trục z có ma trận biến đổi là

$$SH_{xy}(sh_x, sh_y) = \begin{bmatrix} 1 & 0 & sh_x & 0 \\ 0 & 1 & sh_y & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \textbf{(5.33)}$$

Tương tự như vậy chúng ta có ma trận biến đổi của phép kéo (x,z) và phép kéo (y,z). Việc kết hợp các phép biến đổi trong 3D cũng tương tự như trong 2D.

5.2. Phép chiếu

Trong phần này, chúng ta sẽ làm quen với các phép chiếu từ ba chiều vào hai chiều. Một phép chiếu được xác định bởi các **tia chiếu** (*projectors*) đi từ một **tâm chiếu** (*center of projection*) qua các điểm của đối tượng và giao với **mặt phẳng chiếu** (*projection plane*).

Các phép chiếu được chia làm hai loại: **chiếu phối cảnh** (*perspective projection*) và **chiếu song song** (*parallel projection*). Phép chiếu phối cảnh là phép chiếu mà khoảng cách từ tâm chiếu tới mặt phẳng chiếu là hữu hạn. Khi khoảng cách từ tâm chiếu tới mặt phẳng chiếu là vô hạn, hay các tia chiếu song song với nhau, chúng ta có phép chiếu song song.

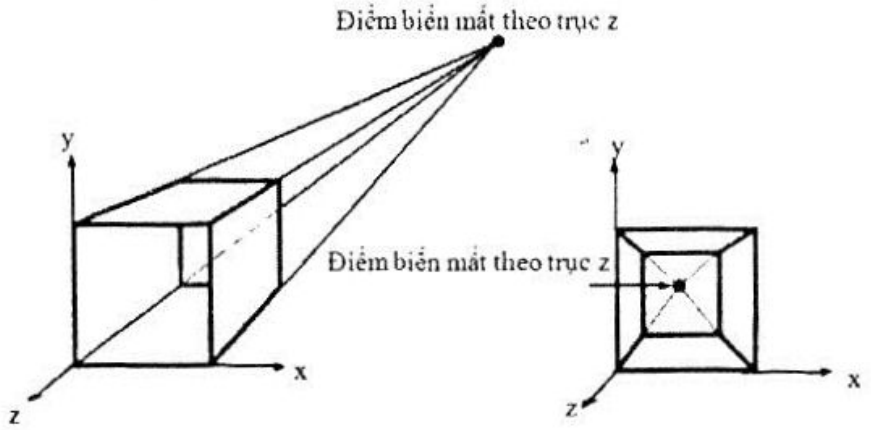
5.2.1. Phép chiếu phối cảnh

Hiệu ứng trực quan của phép chiếu phối cảnh giống như hiệu ứng của hệ thống máy chụp ảnh và hiệu ứng của hệ thống thị giác

con người, và thường được biết đến như là **phép vẽ phối cảnh gần xa** (*perspective foreshortening*): kích thước hình chiếu của một vật thể biến đổi theo tỉ lệ nghịch với khoảng cách từ vật đó đến tâm chiếu. Chính vì lý do đó, mặc dù chiếu phối cảnh cho một cái nhìn thực tế, nó thường không được dùng trong trường hợp cần ghi lại chính xác kích thước, hình dạng và các số đo khác của các vật thể: các góc chỉ được bảo toàn trên những mặt song song với mặt phẳng chiếu, các đường song song thường không còn song song nữa, v.v.

Các ảnh chiếu phối cảnh của một tập các đường thẳng song song với nhau và không song song với mặt phẳng chiếu sẽ hội tụ tại một điểm **biến mất** (*vanishing point*). Trong 3D, các đường thẳng song song chỉ gặp nhau tại vô cùng, do đó, điểm biến mất cũng có thể coi là ảnh chiếu của một điểm ở vô cùng. Và tất nhiên, có vô số điểm biến mất, mỗi điểm ứng với một hướng của các đường thẳng song song nói trên.

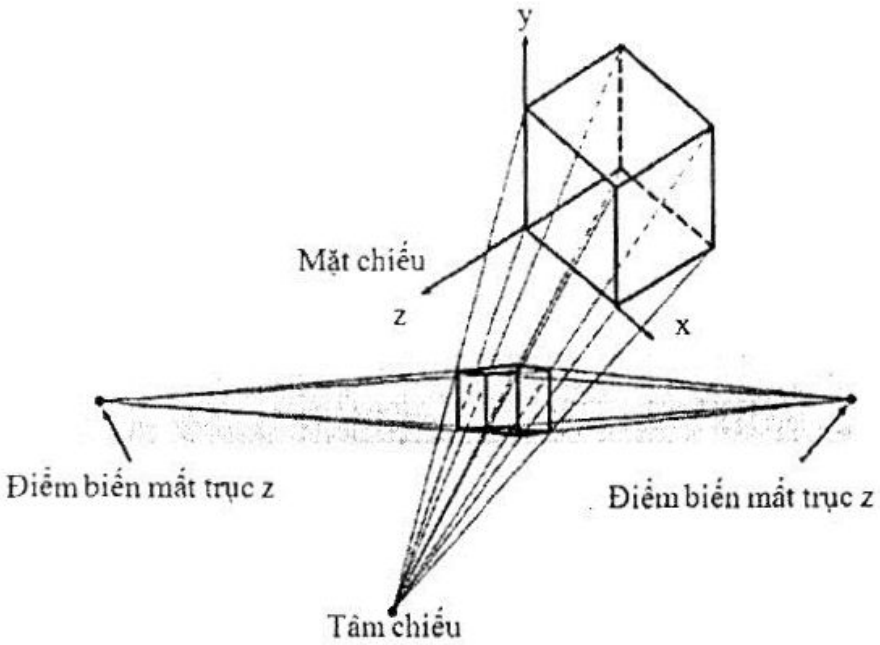
Nếu tập các đường thẳng song song với một trong ba trục tọa độ, điểm biến mất được gọi là **điểm biến mất theo trục** (*axis vanishing point*). Có tối đa ba điểm như vậy, tương ứng với số lượng giao điểm giữa các trục tọa độ và mặt phẳng chiếu.



Hình 5.7. Các phép chiếu một điểm của một hình hộp lên mặt phẳng chiếu không song song với trục z, cho thấy một điểm biến mất theo trục z.

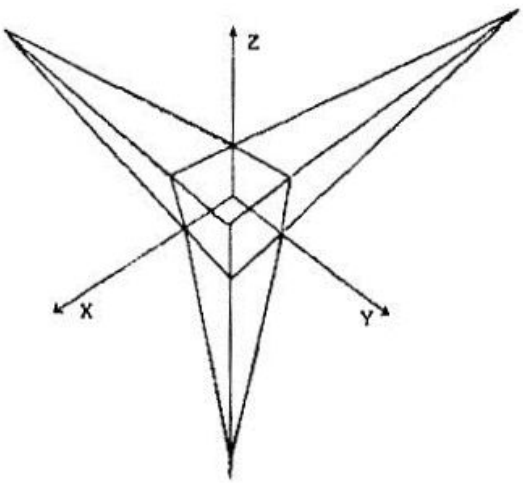
Các phép chiếu phối cảnh được phân loại theo số lượng điểm biến mất theo trục, cũng có nghĩa là theo số lượng giao điểm giữa các trục tọa độ và mặt phẳng chiếu. Hình 5.7 cho thấy hai phép chiếu một điểm khác nhau. Rõ ràng đây là các phép chiếu một điểm vì các đường thẳng song song với trục x và y không hội tụ sau khi

chiếu, chỉ có các đường thẳng song song với z là hội tụ sau khi chiếu.



Hình 5.8. Phép chiếu hai điểm của một hình hộp. Mặt phẳng chiếu cắt trục x và trục z.

Hình 5.8 cho thấy phép chiếu 2 điểm. Lưu ý rằng các đường thẳng song song với trục **y** không hội tụ sau khi chiếu. Phép chiếu phối cảnh hai điểm thường được dùng trong kiến trúc, thiết kế công nghiệp, quảng cáo... Phép chiếu phối cảnh ba điểm thường ít được dùng. Hình 5.9 cho thấy phép chiếu 3 điểm.

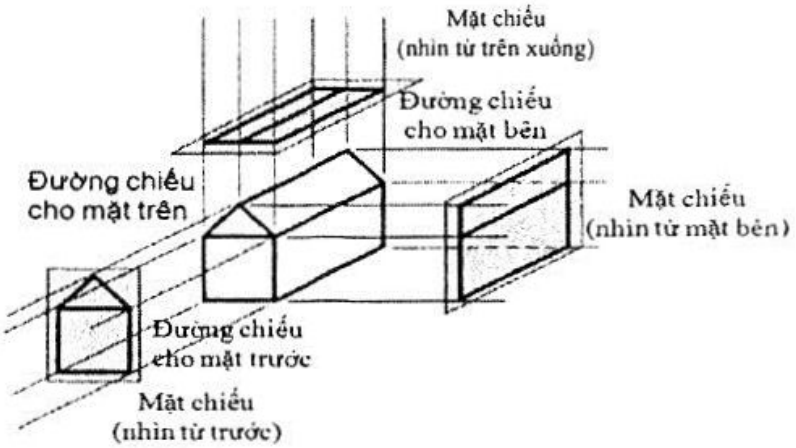


Hình 5.9. Phép chiếu 3 điểm của một hình hộp

5.2.2. *Phép chiếu song song*

Các phép chiếu song song được chia làm hai loại, phụ thuộc vào mối quan hệ giữa hướng chiếu và véc-tơ pháp tuyến của mặt phẳng chiếu. Đối với các **phép chiếu song song trực giao** (*orthographic*), hai hướng này là một (hoặc ngược nhau). Với các **phép chiếu song song xiên** (*oblique*), hai hướng này không trùng nhau.

Có ba loại chiếu trực giao thường dùng là: **chiếu mặt trước** (*front-elevation*), **chiếu mặt trên** (*top-elevation*), và **chiếu mặt bên** (*side-elevation*). Trong cả ba loại chiếu này, mặt phẳng chiếu vuông góc với một trục tọa độ, chính là hướng chiếu. Hình 5.10 cho thấy ví dụ về ba loại chiếu này. Chúng thường được dùng trong kỹ thuật để vẽ các chi tiết máy móc, các tòa nhà vì khoảng cách và các góc có thể tính được từ các hình chiếu này.

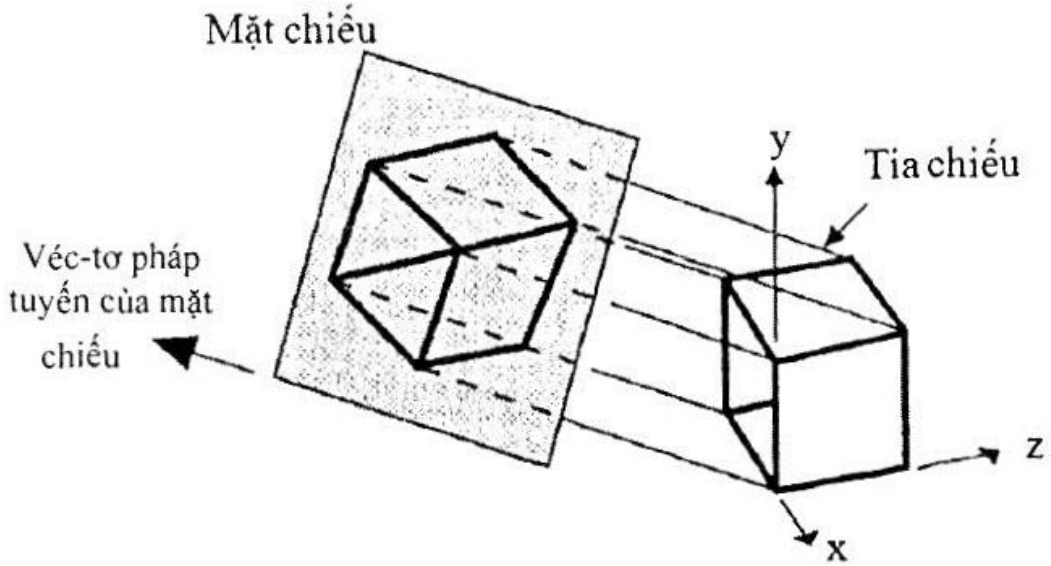


Hình 5.10. Ba phép chiếu trực giao mặt trước, mặt trên, và chiếu mặt bên.

Phép chiếu trực giao có trục đo (*axonometric orthographic projection*) sử dụng mặt phẳng chiếu không vuông góc với trục tọa độ, do đó có thể cho phép nhìn thấy nhiều mặt của vật thể cùng một lúc. Đây là điều mà phép chiếu này giống với phép chiếu phối cảnh, tuy nhiên điểm khác chính là việc vẽ rút gọn gần xa lại là đồng nhất, thay vì phụ thuộc vào khoảng cách đến tâm chiếu. Các đường song song được bảo toàn, tuy nhiên các góc thì không, và khoảng cách có thể được đo theo từng trục (nói chung với những hệ số khác nhau).

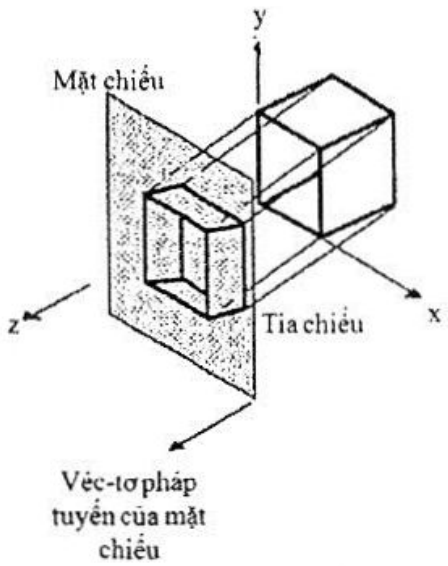
Phép chiếu cùng kích thước (*isometric projection*) là loại phép chiếu có trục đo thường dùng. Véc-tơ pháp tuyến của mặt phẳng chiếu tạo nên với các trục tọa độ các góc bằng nhau. Nếu véc-tơ pháp tuyến của mặt phẳng chiếu là (dx,dy,dz), thì giá trị tuyệt đối của chúng phải bằng nhau, có nghĩa là |dx|=|dy|=|dz|. Chỉ có tám hướng là thỏa mãn điều kiện này. Hình 5.11 cho thấy phép chiếu cùng kích thước dọc theo hướng (1,-1,-1).

Phép chiếu cùng kích thước có một tính chất hữu ích là cả ba trục tọa độ được vẽ rút gọn gần xa với hệ số như nhau. Thêm vào đó, ảnh chiếu của ba trục tọa độ từng cặp tạo ra với nhau những góc 120 độ.



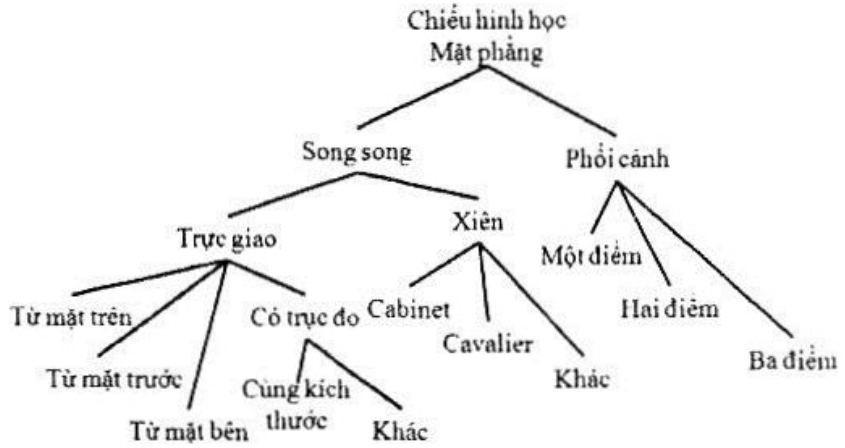
Hình 5.11. Phép chiếu cùng kích cỡ của một hình hộp.

Chiếu xiên (*oblique projection*), lớp chiếu song song thứ hai, khác với chiếu trực giao ở chỗ véc-tơ pháp tuyến của mặt phẳng chiếu không trùng với hướng chiếu. Chiếu xiên kết hợp những tính chất của chiếu trực giao mặt trước, mặt trên và mặt bên với chiếu có trục đo: véc-tơ pháp tuyến của mặt phẳng chiếu trùng với một trục tọa độ, cho phép dùng hình chiếu của những mặt song song với mặt phẳng chiếu để tính góc và khoảng cách. Các mặt khác của vật thể cũng được dùng để đo khoảng cách (không đo được góc) dọc theo các trục tọa độ. Hình 5.12 cho thấy một phép chiếu xiên.



Hình 5.12. Một phép chiếu xiên.

Hình 5.13 cho thấy cách phân loại các phép chiếu.



Hình 5.13. Phân loại các phép chiếu.

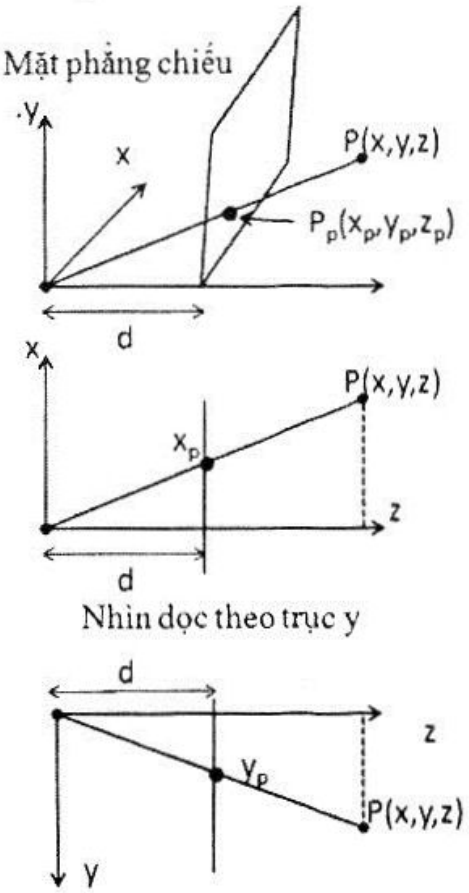
5.2.3. Cơ sở toán học của các phép chiếu

5.2.3.1. Mô tả toán học của phép chiếu phối cảnh

Một phép chiếu chuyển đổi phối cảnh được xác định bởi tâm chiếu và mặt phẳng quan sát. Mặt phẳng quan sát được xác định bởi **điểm nhìn tham chiếu** (*view reference point*) $\mathbf{R_0}$ và pháp tuyến của mặt phẳng quan sát **N**. Một điểm **P** trên đối tượng được xác định trong tọa độ thế giới thực tại vị trí **(x,y,z)**. Công việc ở đây là phải xác định tọa độ điểm ảnh **P'(x',y',z')**.

Với phép chiếu phối cảnh, để đơn giản hóa, chúng ta giả thiết mặt phẳng chiếu vuông góc với trục **z** tại **z = d**, và trong phép chiếu

song song, mặt phẳng chiếu là mặt phẳng **z** = 0. Mỗi phép chiếu có thể được định nghĩa bằng một ma trận 4x4.



Hình 5.14. Chiếu phối cảnh

Với phép chiếu phối cảnh, ta cần chiếu một điểm **P** lên mặt phẳng chiếu **z** = **d** cách tâm chiếu tại gốc tọa độ một khoảng là |**d**|. Để xác định $\mathbf{P_p} = (\mathbf{x_p},\mathbf{y_p},\mathbf{z_p})$, ảnh chiếu phối cảnh của điểm **P** = (**x**,**y**,**z**) lên mặt phẳng chiếu **z** = **d**, sử dụng các tam giác đồng dạng trong Hình 5.14, chúng ta có:

$$\frac{x_p}{d} = \frac{x}{z}; \quad \frac{x_p}{d} = \frac{y}{z}.$$

Nhân hai bên với **d** thu được:

$$x_p = \frac{d \cdot x}{z} = \frac{x}{z/d}. \qquad y_p = \frac{d \cdot y}{z} = \frac{y}{z/d}.$$

Khoảng cách **d** chính là hệ số co giãn áp dụng cho $\mathbf{x_p}$ và $\mathbf{y_p}$. Phép chia cho **z** làm cho ảnh chiếu của vật ở xa nhỏ hơn ảnh chiếu của vật ở gần. Mọi giá trị của **z** đều hợp lệ trừ **z** = **0**.

Phép biến đổi trong công thức trên có thể được thể hiện bằng một ma trận 4x4:

$$M_{per} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 1/d & 0 \end{bmatrix}$$

Nhân điểm $P = \begin{bmatrix} x & y & z & 1 \end{bmatrix}^T$ với ma trận M_{per}, chúng ta thu được điểm $\begin{bmatrix} X & Y & Z & W \end{bmatrix}^T$:

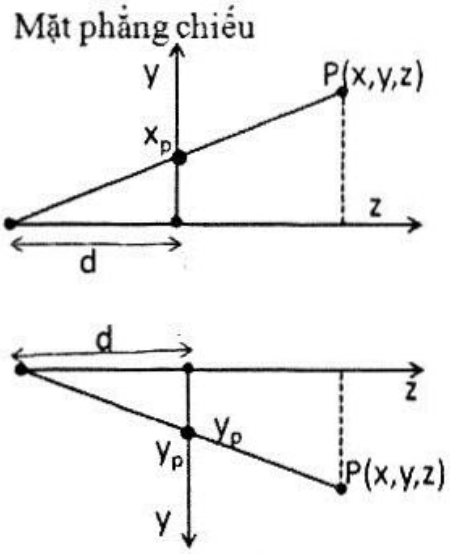
$$\begin{bmatrix} X \\ Y \\ Z \\ W \end{bmatrix} = M_{per} \cdot P = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 1/d & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ x \\ 1 \end{bmatrix}$$

hay

$$\begin{bmatrix} X & Y & Z & W \end{bmatrix}^T = \begin{bmatrix} x & y & z & \frac{z}{d} \end{bmatrix}$$

Chia cho $W(z/d)$, và bỏ tọa độ đồng nhất cuối cùng, chúng ta thu được:

$$\left(\frac{X}{W}, \frac{Y}{W}, \frac{Z}{W}\right) = (x_p, y_p, z_p) = \left(\frac{x}{z/d}, \frac{y}{z/d}, d\right)$$



Hình 5.15. Một phép chiếu phối cảnh khác.

Một công thức khác cho phép chiếu phối cảnh xuất phát từ việc đặt mặt phẳng chiếu tại $z = 0$ và tâm chiếu tại $z = -d$, như trong Hình 5.15. Sự đồng dạng của các tam giác cho chúng ta:

$$\frac{x_p}{d} = \frac{x}{z+d}, \quad \frac{y_p}{d} = \frac{y}{z+d}$$

Nhân cả hai phía với d, chúng ta có:

$$x_p = \frac{d \cdot x}{z+d} = \frac{x}{(z/d)+1}, \qquad y_p = \frac{d \cdot y}{z+d} = \frac{y}{(z/d)+1}$$

Và ma trận là:

$$M_{pcr} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 1/d & 1 \end{bmatrix} \tag{5.34}$$

Công thức này cho phép khoảng cách d có thể tiến đến vô cùng.

Phép chiếu trực giao lên mặt phẳng tại $z = 0$ khá đơn giản. Hướng chiếu trùng với véc-tơ pháp tuyến của mặt phẳng chiếu, chính là trục z. Như vậy, một điểm $P = (x, y, z)$ sẽ được chiếu thành:

$$x_p = x,\ y_p = y,\ z_p = 0$$

Phép chiếu này được thể hiện dưới dạng ma trận:

$$M_{ort} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 \end{bmatrix} \tag{5.35}$$

Lưu ý rằng, khi d trong Công thức 5.34 tiến đến vô cùng, Công thức 5.34 trở thành Công thức 5.35. Điều này là do phép chiếu trực giao là một trường hợp đặc biệt của chiếu phối cảnh.

Câu hỏi và bài tập

1. Hãy nêu ý nghĩa của tọa độ đồng nhất.
2. Thế nào là phép chiếu một điểm, hai điểm và ba điểm?
3. Hãy xác định ma trận của phép đối xứng gương qua mặt phẳng Oxy, Oxz, Oyz?
4. Cho tam giác A(5, 2,1); B(3, 4, 3) và C(1, 1, 10). Hãy xác định tọa độ mới của đỉnh tam giác khi quay một góc 45 độ quanh một trục đi qua điểm P(4, 5, 2) và song song với trục Oy.
5. Tìm ma trận biến đổi trong phép đối xứng qua đường thẳng nằm nghiêng có độ nghiêng là m và đi qua điểm (0, c).
6. **Bài tập lập trình:** Chương trình vẽ - hãy mở rộng chương trình đã phát triển trong bài tập lập trình ở chương trước để cho phép người sử dụng chọn một đoạn thẳng đã vẽ và thao tác với nó qua các phép biến đổi: quay, tịnh tiến, và co giãn.

Chương 6
MÔ HÌNH HÓA ĐỐI TƯỢNG

Có hai nhiệm vụ chính trong việc tạo ra bức ảnh của một cảnh ba chiều là mô hình hóa và kết xuất đồ họa. Việc mô hình hóa phải tạo ra mô hình - sự mô tả các đối tượng - sẽ được dùng trong hệ thống đồ họa. Các mô hình phải được tạo ra cho mọi đối tượng trong cảnh vật; chúng phải thể hiện được chính xác hình dạng và diện mạo của đối tượng. Việc kết xuất đồ họa, lấy các mô hình làm đầu vào và tạo ra các giá trị điểm ảnh cho ảnh cuối cùng. Một vật thể đơn giản, như một cái hộp, có thể được hiện thông qua các đa giác cho các mặt của hộp. Nhiệm vụ đặt ra là phải làm thế nào xác định được các tọa độ 3 chiều của mỗi đỉnh đa giác tạo nên mô hình. Để có thể kết xuất chính xác diện mạo của một đối tượng, chúng ta còn phải xác định giá trị màu sắc, véc-tơ pháp tuyến tạo bóng, và tọa độ ảnh **chất liệu** (*texture*) cho các điểm và mặt của mô hình.

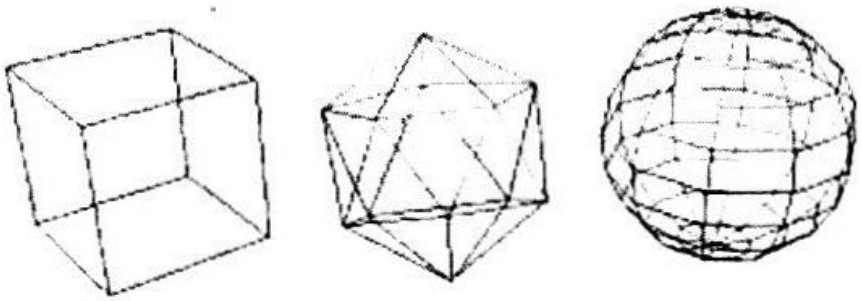
Các chương trước đã mô tả một số nền tảng toán học và thuật toán đồ họa máy tính cần thiết để hiển thị các điểm và các đoạn hai và ba chiều. Như vậy, chúng ta đa có thể tạo ra các hệ thống đồ họa có thể xử lý các đối tượng phức tạp được thể hiện dưới dạng các cạnh (chế độ khung dây - wireframe). Với một số trường hợp, thể hiện các đối tượng dưới dạng khung dây là đủ. Mặt khác, các đối tượng được hiển thị ở chế độ này chưa chân thực và bắt mắt. Để có thể hiển thị các đối tượng chân thực hơn, chúng ta còn cần phải mô hình hóa đối tượng chi tiết hơn. Các mô hình này phải thể hiện được hình dạng và vẻ bề ngoài của các đối tượng. Ngoài ra, chúng ta cũng phải đưa ra các phương pháp lưu trữ hiệu quả các mô hình này.

6.1. Vẽ kỹ thuật

Các phương pháp mô hình hóa đối tượng liên tục thay đổi theo thời gian. Những thay đổi bắt đầu từ trước khi máy tính xuất hiện và người ta chỉ sử dụng bút chì và giấy. Vẽ kỹ thuật là một trong những phương pháp đầu tiên trong việc mô hình hóa các đối tượng. Kỹ thuật này không liên quan gì đến máy tính mà nó chỉ là phương tiện liên lạc giữa con người. Phương pháp này có thể có dẫn đến những sai sót, tuy nhiên với những cảm nhận chung, con người vẫn có thể hiểu chính xác đối tượng cần mô tả. Ngoài ra, cũng không có định nghĩa chính thức nào về vẽ kỹ thuật như một phương pháp biểu diễn đối tượng. Ý tưởng đơn giản của kỹ thuật này là thể hiện các vật thể dưới các phép chiếu phẳng. Tất nhiên đây là một phương pháp biểu diễn khá nhập nhằng vì nếu một người tìm cách cài đặt trên máy tính, rất khó để xác định xem bao nhiêu phép chiếu hai chiều là đủ để thể hiện đầy đủ một vật thể ba chiều.

6.2. Thể hiện khung dây

Thể hiện **khung dây** (*wireframe*) là cách thể hiện đầu tiên đối với các vật thể nhiều mặt trên máy tính. Cách làm tự nhiên này là biểu diễn các vật thể chỉ bằng các cạnh của chúng. Có thể nói, các cạnh là một trong những đặc trưng quan trọng nhất của một vật thể mà chúng ta có thể nhìn thấy. Đối tượng được chiếu lên trên màn hình máy tính bằng cách vẽ những đoạn thẳng tại vị trí các cạnh của đối tượng và chọn lựa xóa các đường ẩn thông qua các mặt phẳng cắt ngang. Hình 6.1 cho thấy một ví dụ về biểu diễn các đối tượng thông qua mô hình khung dây.



Hình 6.1. Ví dụ về biểu diễn các đối tượng thông qua mô hình khung dây

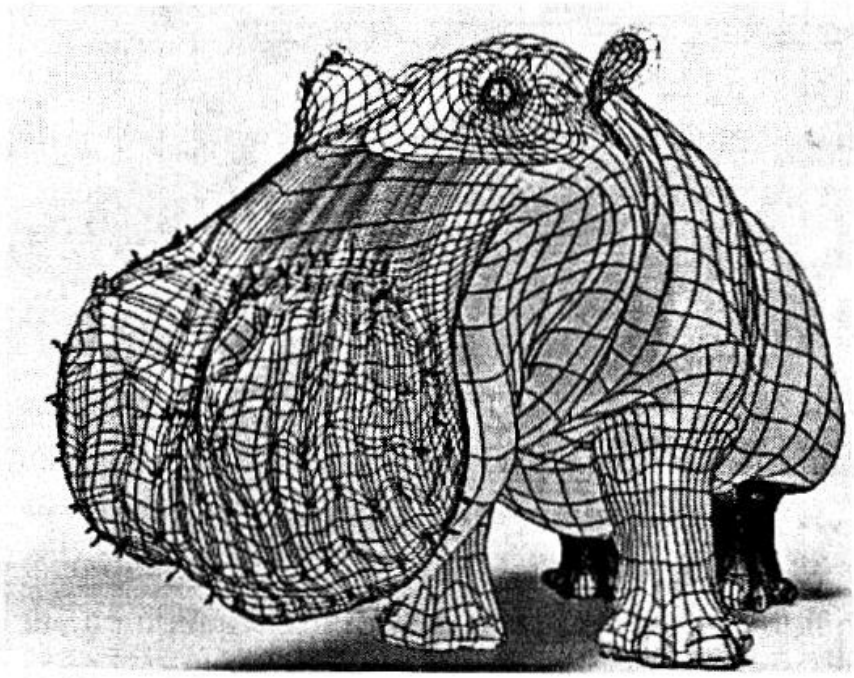
Việc sử dụng mô hình khung dây cho phép chúng ta hình dung được kết cấu bên trong của một mô hình ba chiều. Do việc tính toán để biểu diễn các khung lưới tương đối đơn giản và nhanh chóng, mô hình dạng này thường được sử dụng trong những trường hợp đòi hỏi tỷ lệ số hình ảnh vẽ trên màn hình cao, ví dụ như khi làm việc với một mô hình 3D rất phức tạp hoặc trong các hệ thống thời gian thực. Khi có đòi hỏi cao hơn về chất lượng hiển thị đồ họa, các chất liệu bề mặt có thể được được tự động thêm vào, sau khi đã hoàn thành việc kết xuất ra dạng biểu diễn khung dây của vật thể. Phương pháp này cho phép các nhà thiết kế nhanh chóng xem qua các thay đổi, hoặc xoay chuyển đổi tượng đến một góc nhìn mong muốn, mà không bị sự chậm trễ của quá trình kết xuất ảnh hưởng. Nhược điểm của biểu diễn khung dây là không cho phép người sử dụng hình dung được toàn bộ chi tiết của vật thể đang được biểu diễn.

6.3. Thể hiện bề mặt thông qua đa giác

Như chúng ta đa biết, với cấu trúc mành của các thiết bị hiển thị, các thuật toán dựa trên đường quét làm việc rất hiệu quả với các điểm, các cạnh và các đa giác. Vì vậy, cách mô hình hóa thông dụng và hiệu quả nhất trên các thiết bị hiển thị hiện nay là chính mô hình đa giác. Với mô hình này, các đối tượng được mô tả thông qua các bề mặt đa giác chứ không chỉ thông qua các cạnh.

Mọi đối tượng có thể được xấp xỉ bằng các bề mặt đa giác. Ví dụ, một khối hộp có thể được thể hiện thông qua 6 mặt chữ nhật. Lưu ý rằng đây là cách thể hiện vỏ ngoài của các đối tượng nhằm phục vụ cho việc hiển thị bề ngoài của đối tượng. Các đối tượng phức tạp với các bề mặt cong cũng có thể xấp xỉ thông qua các đa giác. Một bề mặt cong thường được xấp xỉ bằng một lưới các đa giác mà đỉnh của các đa giác này nằm trên bề mặt cong. Ví dụ, một hình cầu có thể được xấp xỉ thông qua một số lượng nhất định các đa giác. Thậm chí, các đối tượng phức tạp, ví dụ như con hà mã trong Hình 6.2, cũng có thể được xấp xỉ thông qua các đa giác. Với cách xấp xỉ bề mặt cong như vậy, mặc dù các đỉnh của đa giác nằm trên bề mặt cong, các điểm trong đa giác không nhất thiết phải nằm

trên bề mặt cong. Chính vì vậy, nếu chỉ hiển thị các mô hình đa giác một cách thông thường, chúng ta sẽ thấy các mô hình này có rất nhiều góc cạnh và chưa có đủ độ cong cần thiết. Khi tăng số lượng đa giác dùng để mô hình một đối tượng có các bề mặt cong lên, ta có thể có những đối tượng chân thực hơn, tuy nhiên giá phải trả là tốn bộ nhớ lưu trữ và thời gian tính toán hơn. Ngoài ra, người ta có thể tạo ra độ chân thực cho các đối tượng được mô hình hóa với các bề mặt đa giác bằng cách áp dụng các kỹ thuật tạo bóng khác nhau.

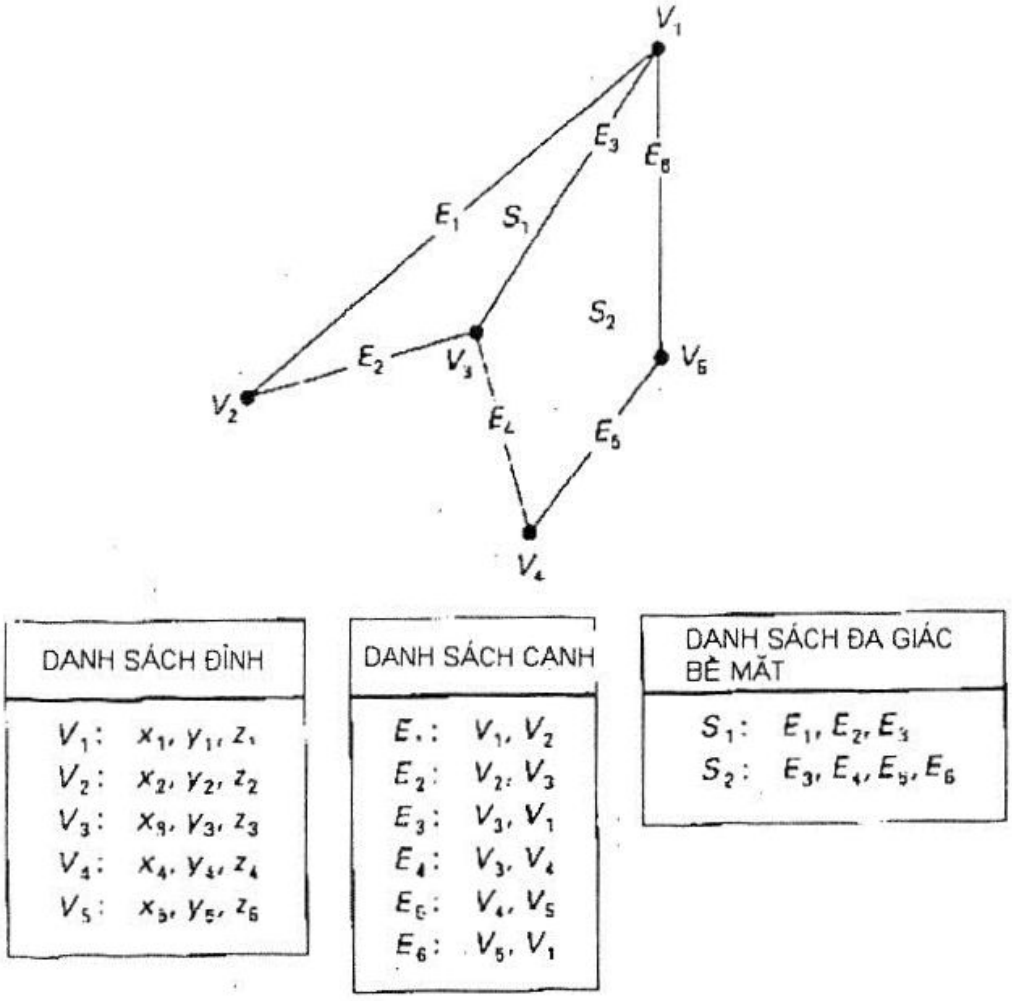


Hình 6.2. Mô hình hóa một con hà mã bằng lưới đa giác

Thông qua các bề mặt đa giác, các vật thể được mô tả bằng **lưới đa giác** (*polygon mesh*). Lưới đa giác là một tập hợp các đa giác, hay các mặt, kết hợp với nhau để tạo thành lớp vỏ của một đối tượng. Đây đã trở thành một cách biểu diễn thông thường để thể hiện một lớp lớn các hình khối đặc trong đồ họa.

Lưới đa giác được lưu trữ theo nhiều cách khác nhau bằng cách lựa chọn việc lưu dữ liệu là đỉnh, cạnh hay mặt đa giác. Khi các đối tượng được mô hình sử dụng đa giác, các đa giác kề nhau có thể có chung cạnh. Để đảm bảo rằng khi kết xuất không có các khoảng trống được tạo ra giữa các đa giác liền kề do quá trình tính toán số

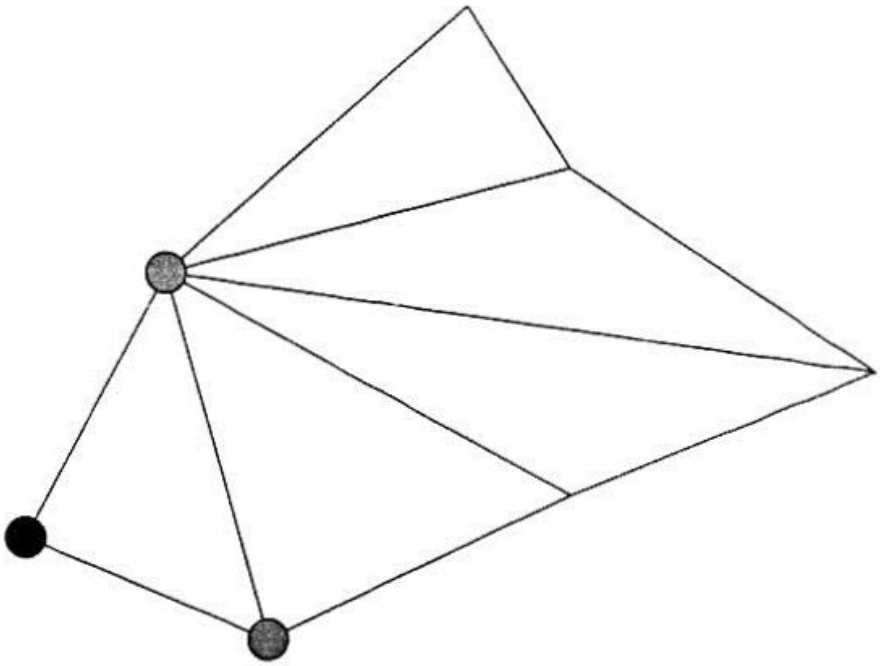
thực các đa giác có cạnh chung nên dùng cùng giá trị tọa độ cho hai đầu mút của các cạnh chung. Chính vì thế, người ta thường dùng cấu trúc chứa các cạnh dữ liệu tham chiếu đến các điểm để vừa tiết kiệm bộ nhớ, vừa giải quyết vấn đề cạnh chung và điểm chung. Hình 6.2 cho thấy cấu trúc dữ liệu để lưu trữ mô hình 3D thông qua lưu trữ danh sách đỉnh, cạnh và đa giác bề mặt. Thông thường, người ta cũng có thể bỏ qua danh sách cạnh và chỉ lưu trữ danh sách đỉnh và đa giác bề mặt.



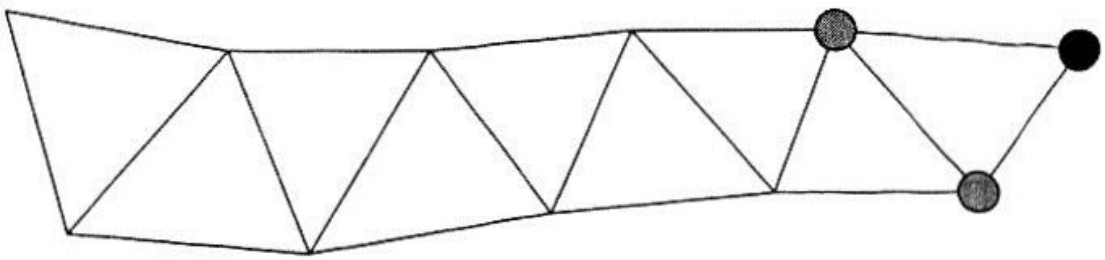
Hình 6.3. Lưu trữ mô hình 3D thông qua lưu trữ danh sách đỉnh, cạnh và đa giác bề mặt.

Một trong những cách đơn giản nhất để tăng tốc một chương trình đồ họa máy tính, đồng thời tiết kiệm được không gian lưu trữ là bến các tam giác hay đa giác độc lập nhau thành một **quạt tam giá**: (*triangle fan*) (xem Hình 6.4) hay một **chuỗi tam giác**

(*triangular strip*) (xem Hình 6.5). Như vậy ngoài tam giác đầu tiên, mỗi tam giác mới chỉ cần lưu trữ thêm một đỉnh. Đồng thời, các phép biến đổi trên các tam giác cũng chỉ cần thực hiện trên một đỉnh, ngoại trừ tam giác đầu tiên.



Hình 6.4. Quạt tam giác.



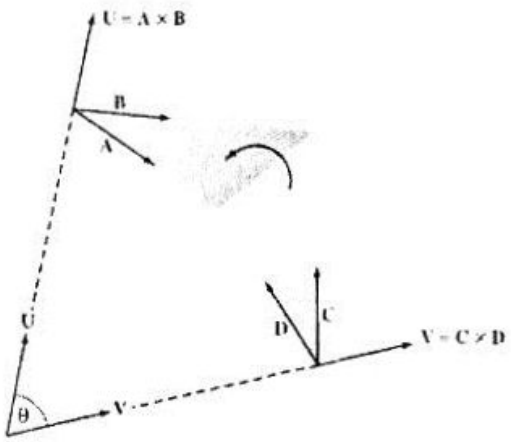
Hình 6.5. Chuỗi tam giác.

6.4. Tạo lưới và phân tách

Tạo lưới (*tessellation*) là quá trình phân tách một bề mặt phức tạp, các đa giác nhiều cạnh thành các đối tượng đơn giản hơn như tam giác hoặc tứ giác (trên 3 chiều). Tam giác thường được sử dụng nhiều hơn vì nó nằm trên một mặt phẳng và có thể mành hóa dễ dàng. Quá trình tạo lưới thường được thực hiện ở khâu tiền xử lý và được lưu trữ lại trước khi đưa vào luồng xử lý đồ họa nhằm tăng hiệu quả của kết xuất đồ họa. Thông qua thuật toán **phân tách** (*decomposition*), các đa giác thường được chuyển một cách đơn

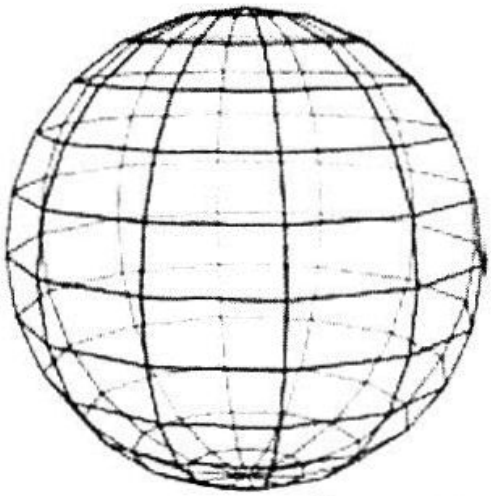
giản thành các quạt tam giác bằng cách giữ một đỉnh làm đỉnh chung của mọi tam giác, đỉnh đó và 2 đỉnh tiếp theo thành một tam giác (xem Hình 6.4). Thuật toán phân tách đơn giản này được thực hiện để tối thiểu hóa tính toán trong quá trình kết xuất. Một cách tiếp cận khác là thực hiện phân tách để tăng chất lượng kết xuất. Vì quá trình tạo bóng giả thiết các đối tượng đơn giản là dẹt, chúng ta phải chọn cách phân tách để tạo ra các tam giác xấp xỉ tốt nhất độ cong của bề mặt sẽ đưa ra kết quả tạo bóng tốt hơn.

Phân tách một tứ giác liên quan đến việc chọn đường chéo nào trong hai đường chéo để phân tách. Một phương pháp để tìm đường chéo để tạo ra độ cong cần thiết là so sánh các góc tạo bởi các véc-tơ pháp tuyến tại hai đỉnh của đường chéo. Góc này đánh giá sự thay đổi về véc-tơ pháp tuyến từ một góc sang góc đối diện. Cặp góc tạo nên sự khác biệt góc nhỏ nhất (gần phẳng nhất) là ứng viên đường chéo tốt nhất; nó tạo ra hai tam giác với sự chênh lệch độ phẳng nhỏ nhất có thể, như trong Hình 6.6. Thuật toán này có thể được thực hiện bằng cách tính tích vô hướng giữa các cặp véc-tơ pháp tuyến, và chọn cặp có tích vô hướng lớn nhất (góc nhỏ nhất). Nếu chúng ta không có véc-tơ pháp tuyến của bề mặt, véc-tơ pháp tuyến tại một điểm có thể được tính bằng cách lấy tích hữu hướng của hai véc-tơ cạnh xuất phát từ điểm đó. Sử dụng độ cong của bề mặt là một cách để đánh giá quá trình phân tách. Ngoài ra, người ta có thể chia tứ giác thành tam giác sao cho chúng có diện tích gần nhau nhất.



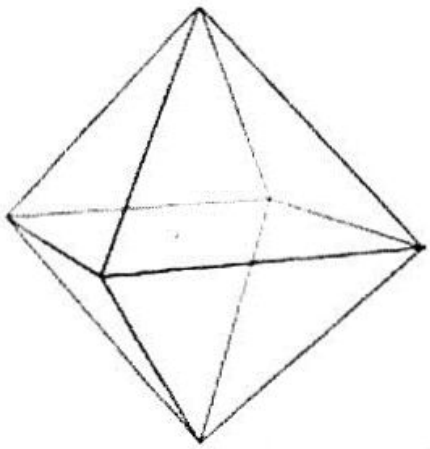
Hình 6.6. Phân tách tứ giác thành hai tam giác.

Tạo lưới cho các bề mặt đơn giản như hình cầu hay hình ống không khó. Phương pháp đơn giản nhất cho hình cầu là tạo lưới theo kinh độ và vĩ độ. Mặc dù thuật toán này dễ cài đặt, nó có bất lợi là các tứ giác được sinh ra có kích thước rất khác nhau, như trong Hình 6.7. Sự khác nhau về kích thước này có thể tạo ra những thành phần lạ rất rõ khi kết xuất, đặc biệt là khi vật thể được chiếu sáng và quay.



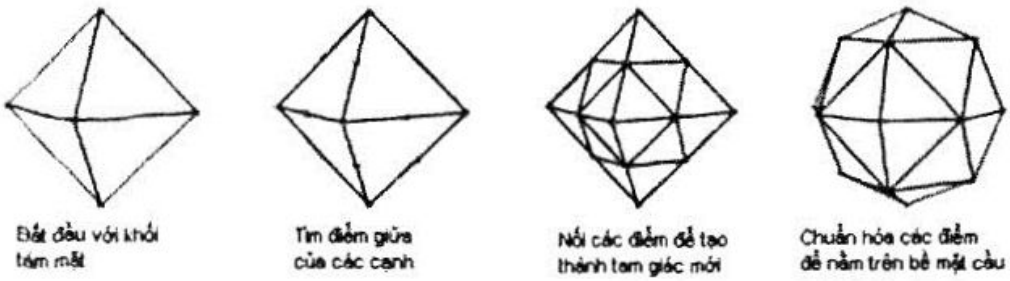
Hình 6.7. Tạo lưới cho bề mặt hình cầu

Thuật toán tạo lưới theo **khối tám mặt** (*octahedral*) và **khối hai mươi mặt** (*icosahedral*) làm việc tốt hơn vì nó tạo ra các tam giác có kích thước gần bằng nhau. Thuật toán này cũng rất dễ cài đặt. Cách tạo lưới theo khối tám mặt xuất phát bằng việc xấp xỉ một hình cầu với một khối tám mặt với các đỉnh nằm trên hình cầu, như trong Hình 6.8. Vì mỗi mặt của khối tám mặt là một tam giác, nó có thể được dễ dàng chia ra thành bốn tam giác mới.



Hình 6.8. Xấp xỉ hình cầu bằng một khối tám mặt

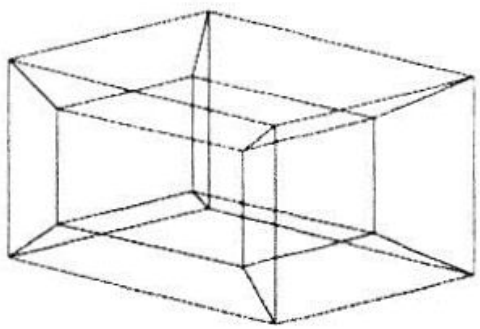
Mỗi tam giác được chia ra bằng cách tạo ra một đỉnh mới ở giữa các cạnh của tam giác sẵn có. Các đỉnh mới này được nối với nhau để tạo ba cạnh mới, và kết quả là bốn tam giác mới được tạo ra từ tam giác ban đầu, như trong Hình 6.9. Để chuẩn hóa tọa độ của các đỉnh mới, người ta chia chúng cho khoảng cách từ đỉnh đó đến gốc tọa độ. Quá trình này làm cho các đỉnh mới nằm trên bề mặt của hình cầu đơn vị. Quá trình này có thể được thực hiện tiếp tục để chia ra thành các tam giác nhỏ hơn.



Hình 6.9. Chia mỗi tam giác của một khối tam mặt ra làm bốn tam giác con

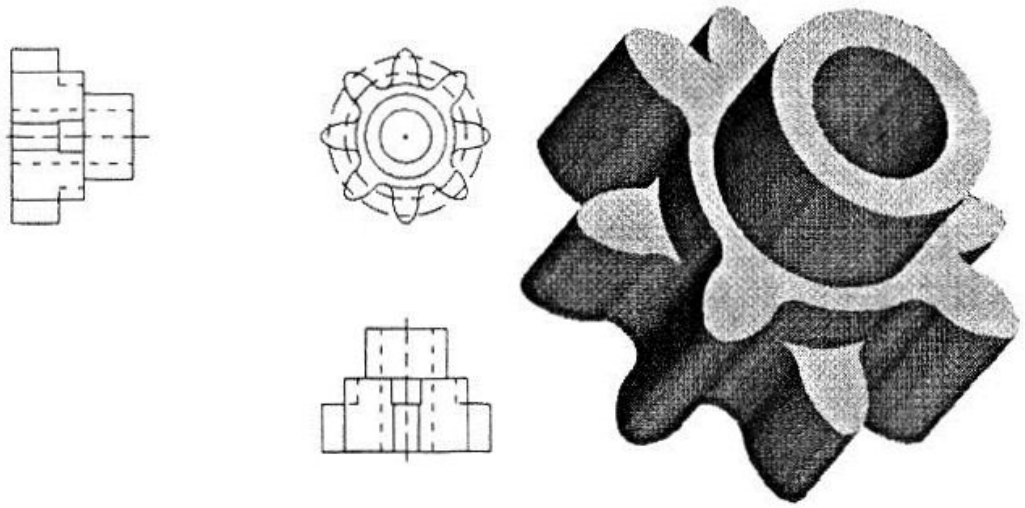
6.5. Mô hình khối rắn

Nhược điểm của thể hiện khung dây là chúng có thể tạo ra sự nhập nhằng. Ví dụ trong Hình 6.10, một hình khối chữ nhật với một lỗ hổng ở trong. Tuy nhiên, chúng ta không thể phân biệt được lỗ hổng đó rỗng theo chiều nào. Chính vì lý do này, người ta còn hướng đến những hình thức mô hình hóa khác toàn diện hơn. Mô hình **khối rắn** (*solid*) là một phương pháp biểu diễn một vật thể ba chiều hoàn chỉnh nhất. Nó có cả diện tích, khối lượng và thể tích.

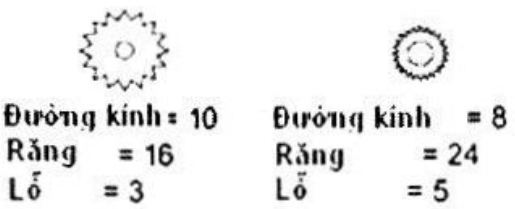


Hình 6.10. Sự nhập nhằng tạo ra bởi thể hiện khung dây.

Mô hình khối rắn có thể được tạo ra bằng cách **quét chụp** (*scan*) đối tượng bằng thiết bị chuyên dụng, **quét cong** (*sweep*) các phác thảo hai chiều (ví dụ như trong Hình 6.11), hoặc từ các lệnh vẽ khối rắn cơ sở (ví dụ như trong Hình 6.12).

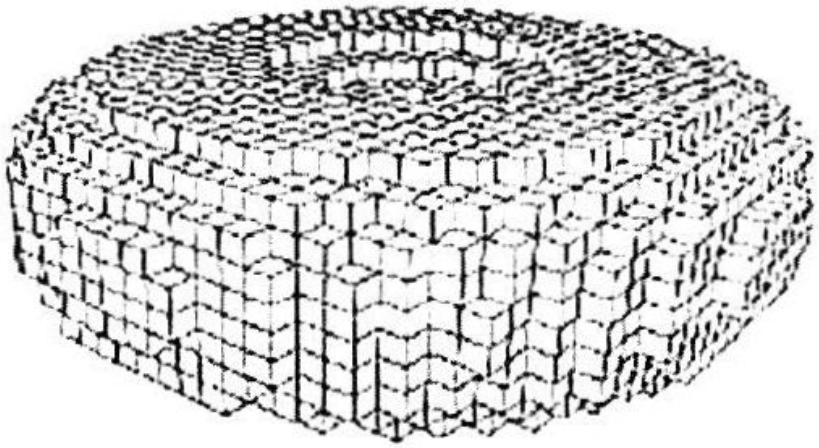


Hình 6.11. Đối tượng ba chiều được mô hình hóa thông qua quét cong phác thảo hai chiều.

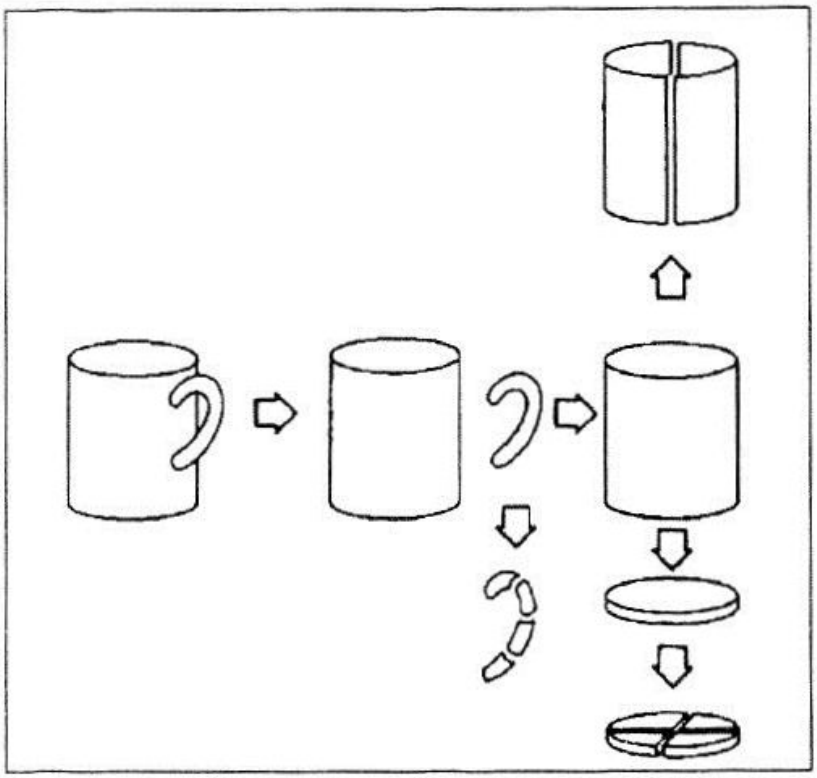


Hình 6.12. Các lệnh vẽ khối rắn cơ sở với các tham số có thể thay đổi

Người ta có thể mô hình hóa một đối tượng thông qua việc liệt kê không gian bao phủ (xem Hình 6.13). Toàn bộ không gian bảo phủ bởi đối tượng được chia nhỏ thành các khối nhỏ (thường là hình lập phương) giống nhau. Mỗi đối tượng sẽ được biểu diễn bởi một tập các khối này. Việc liệt kê không gian bao phủ cũng có thể được thực hiện thông qua việc chia không gian phủ thành các khối không giống nhau mà cũng không được tạo mẫu trước. Hình dạng và kích thước của các các khối sẽ tùy thuộc vào mô hình cần tạo dựng (xem Hình 6.14).

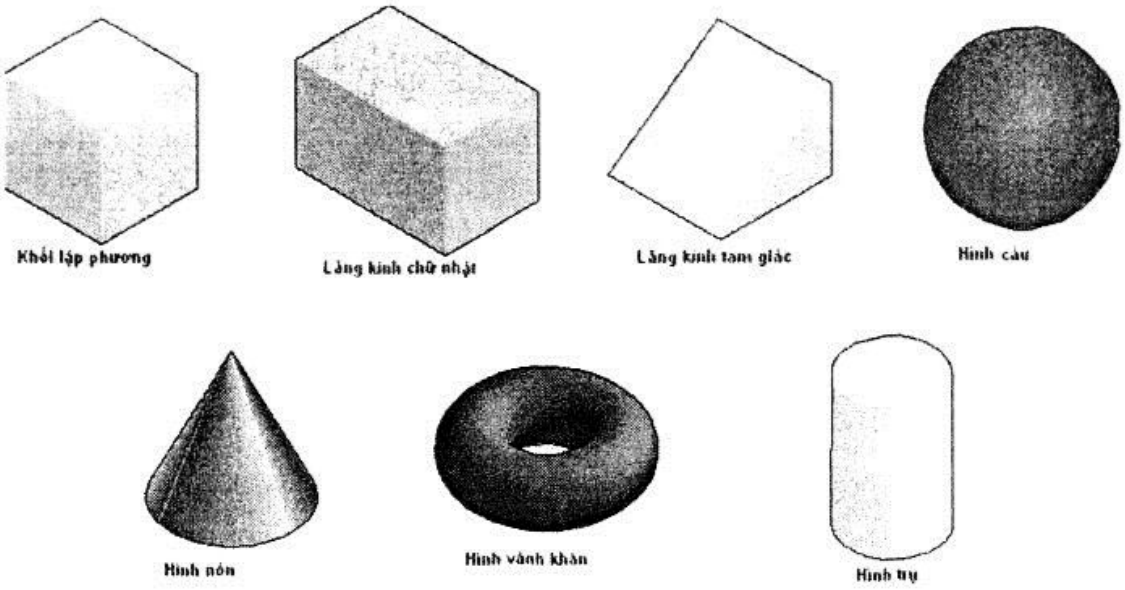


Hình 6.13. Vật thể được mô hình hóa thông qua liệt kê không gian bao phủ



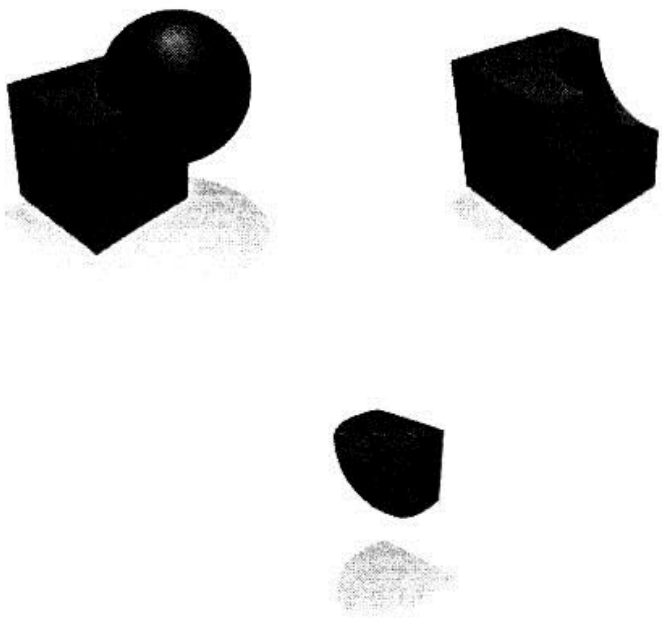
Hình 6.14. Chia không gian bao phủ thành các khối không giống nhau

Một phương pháp cũng thường được sử dụng là **mô hình khối rắn một cách xây dựng** (*constructive solid geometry - CSG*). Trong phương pháp này, người ta sử dụng các mẫu cơ bản cũng là các mô hình khối rắn và các toán tử logic để tạo dựng nên các bề mặt hoặc đối tượng phức tạp từ những mẫu này. Các mẫu cơ bản thường là các khối hộp, khối trụ, khối lăng trụ, khối chóp, khối cầu, khối nón, ... như trong Hình 6.15.



Hình 6.15. Các hình khối cơ sở thường được sử dụng trong mô hình khối rắn.

Các toán tử logic thường dùng là: phép hợp, phép giao và phép loại trừ như trong Hình 6.16.



Hình 6.16. a) Phép hợp của hai đối tượng; b) Phép loại trừ; c) Phép giao.

Câu hỏi và bài tập

1. Hãy nêu ưu điểm của cách lưu trữ bằng quạt tam giác và chuỗi tam giác.
2. Người ta thường biểu diễn các đối tượng phức tạp trong đồ họa máy tính như thế nào?
3. Hay nêu nhược điểm của mô hình khung lưới khi biểu diễn đối tượng.
4. **Bài tập lập trình:** Chương trình vẽ - hãy mở rộng chương trình đã phát triển trong bài tập lập trình ở chương trước thành ứng dụng vẽ 3D sử dụng OpenGL. Chương trình này sẽ cho phép người sử dụng nạp và hiển thị một số hình khối từ một tệp dữ liệu văn bản có cấu trúc như sau:

```
Dòng đầu tiên chứa số N là số lượng đỉnh của
hình khối.
N dòng tiếp theo, mỗi dòng chứa 3 số là tọa độ
của N đỉnh.
Dòng tiếp theo chưa M là số lượng mặt tam giác
của hình khối.
M dòng tiếp theo, mỗi dòng chứa 3 số nguyên là
số hiệu đỉnh của từng tam giác trong M tam
giác.
```

Chương 7
XÁC ĐỊNH MẶT HIỆN

Sau khi dựng nên các khối hình học, bước tiếp theo trong quá trình kết xuất chân thực một khung cảnh từ một điểm nhìn cho trước là xác định xem ta có thể nhìn thấy những vùng bề mặt nào từ điểm nhìn. Đây là bài toán **Xác định mặt hiện.** Trước đây, bài toán này được gọi là **Loại bỏ mặt khuất**, nhưng sau đó được thay đổi để nhấn mạnh đến những mặt tích cực hơn là những mặt tiêu cực của bài toán. Trong nhiều năm qua đã có nhiều thuật toán xác định mặt hiện được xây dựng. Chương này đề cập đến những thuật toán nổi trội.

Nói chung, những thuật toán xác định mặt hiện có thể được phân loại thành 3 dạng sau: chính xác theo đối tượng (*object precision*), chính xác theo ảnh (*image precision*) hoặc ưu tiên theo danh sách (*list priority*).

Các thuật toán chính xác theo đối tượng được mô tả như sau:

```
Với mỗi đối tượng O trong thực tại
      Tìm phần A của O có thể nhìn thấy
      Hiển thị A một cách tương đối
```

trong khi đó các thuật toán chính xác theo ảnh lại là:

```
Với mỗi điểm ảnh trên màn hình thì
      Xác định vị trí điểm ảnh mà đối tượng
      hiện O có bị tia chiếu từ điểm nhìn (máy
      quay hoặc mắt) chạm tới.
      Nếu có đối tượng O như thế thì hiển thị
      điểm ảnh này với màu phù hợp
      Nếu không thì hiển thị điểm ảnh với màu
      nền.
```

Cả 2 dạng thuật toán này đều thực hiện tất cả các phép tính với độ chính xác theo những đặc trưng (đối tượng hay là ảnh) mà thuật toán căn cứ vào. Khác biệt chính giữa chúng là thuật toán chính xác theo đối tượng tính toán chính xác tất cả những đối tượng thành phần có thể nhìn thấy được trong khung cảnh trong khi thuật toán chính xác theo ảnh lại xác định những thành phần này chỉ trong một số chiều lấy mẫu nhất định. Tính phức tạp của các thuật toán chính xác theo ảnh phụ thuộc vào độ phân giải màn hình, trong khi tính phức tạp của thuật toán chính xác theo đối tượng lại không phụ thuộc vào yếu tố này.

Như vậy, thuật toán chính xác theo đối tượng phải chú ý hiện tượng răng cưa. Các thuật toán thuần túy chính xác theo đối tượng thường chỉ được sử dụng trong thời kỳ đầu của đồ họa, chủ yếu là trên các thiết bị đồ họa vec-tơ.

Kỹ thuật dò tia mang ý tưởng của các thuật toán chính xác theo ảnh, được phân loại thành các thuật toán xử lý theo vùng hay theo điểm. Thuật toán Warnock là một minh họa điển hình cho các thuật toán xử lý theo vùng. Thuật toán bộ đệm Z, thuật toán Watkins và kỹ thuật dò tia lại minh họa cho các thuật toán xử lý theo điểm.

Các thuật toán ưu tiên theo danh sách đứng trung gian giữa 2 kiểu chính xác theo đối tượng và chính xác theo ảnh. Chúng khác các thuật toán chính xác theo ảnh ở chỗ chúng tính toán trước trong không gian đối tượng, thứ tự hiện của các mặt trước khi quét chuyển các đối tượng vào không gian ảnh theo thứ tự từ sau ra trước. Để thu được thứ tự này, người ta có thể phải chia nhỏ các đối tượng. Thuật toán Schumacker, Newell-Newell-Sancha và cây BSP thuộc loại này.

Giống như trong Chương 4, mỗi thuật toán trình bày trong chương này đã được cân nhắc lựa chọn, đáp ứng các yêu cầu:

(1) Thuật toán đó là một trong những thuật toán tốt nhất của một loại nào đó tại thời điểm hiện tại;
(2) Thuật toán đó có ý nghĩa về mặt lịch sử và dễ mô tả;
(3) Thuật toán đó liên quan đến các kỹ thuật đáng chú ý.

Với các tiêu chí này, các thuật toán được phân loại như sau:

Thuật toán	Tiêu chí	Chú giải
Schumacker-Brand-Gilliland-Sharp	(2)	Thuật toán ưu tiên danh sách đầu tiên.
Newell-Newell-Sancha	(2), (3)	Một thuật toán ưu tiên danh sách sắp xếp theo chiều sâu.
BSP	(1), (2)	Một thuật toán ưu tiên danh sách.
Warnock	(2), (3)	Một thuật toán chính xác theo ảnh chia nhỏ từng khu vực
Weiler-Atherton	(2), (3)	Một thuật toán chia nhỏ từng khu vực ở mức phức tạp hơn
Z-buffer	(1)	Thuật toán này đầu tiên chỉ được sử dụng trên các trạm làm việc đồ họa cao cấp nhưng hiện nay nó được áp dụng cho hầu hết các hệ thống đồ họa.
Watkins	(1), (2)	Một thuật toán dòng quét, vẫn là sự lựa chọn thích hợp trong các trình ứng dụng hiện nay vì nó nhanh hơn các thuật toán dạng dò tia.
Dò tia	(1)	Kỹ thuật này được áp dụng cùng với các phương thức tính độ va đập của ánh sáng (radiosity) nếu ta muốn thu được những bức ảnh có độ chân thực cao nhất.
Octree	(1)	Một thuật toán ưu tiên danh sách được dùng nhiều trong kết xuất hình khối.
Thuật toán bề mặt cong Blinn	(2), (3)	

Một trong những thuật toán trên sẽ được đề cập từ mục 7.3 trở đi. Mục 7.2 sẽ mô tả một bước tiền xử lý được gọi là bước loại bỏ mặt quay vào trong. Bước này không khó để thực hiện và có thể dùng để loại bỏ nhiều thao tác thừa. Trước khi đi tiếp, chúng ta sẽ thống nhất với nhau với số giả thiết nhằm thuận tiện cho việc mô tả. Trừ những trường hợp được quy định rõ ràng, ta sẽ giả thiết:

> ***Phép chiếu trực giao**: Ta giả sử rằng thế giới xung quanh được biến đổi sang hệ tọa độ thông qua phép chiếu trực giao mà tâm chiếu nằm ở vô cùng và tương ứng là loại bỏ một cách đơn giản tọa độ z.*

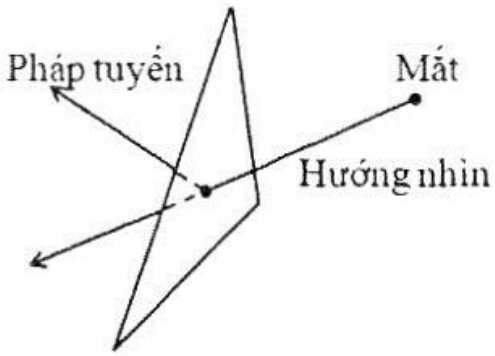
Cũng phải nói thêm rằng có cả những thuật toán **Xác định đoạn thẳng hiện** hay thuật toán **Loại bỏ đoạn thẳng khuất**. Những thuật toán này được sử dụng chủ yếu trong hiển thị nội dung dạng **khung lưới** (*wireframe*). Tất cả các thuật toán xác định mặt hiện đều có thể sửa đổi thành thuật toán xác định đoạn thẳng hiện, tuy nhiên điều ngược lại thì không thực hiện được. Thuật toán xác định đoạn thẳng hiện đầu tiên được Roberts xây dựng trong [Robe63]. Thuật toán này đặt giả thiết tất cả các đoạn thẳng xuất phát từ cạnh của các đa diện lồi. Đầu tiên, các cạnh quay vào trong sẽ bị loại bỏ (một cạnh được coi là quay vào trong khi nó là cạnh chung của hai mặt quay vào trong). Từng cạnh còn lại sẽ được kiểm tra xem có đa diện nào che khuất nó. [Roge98] mô tả thuật toán này chi tiết hơn. Một thuật toán xác định đường thẳng hiện đa năng hơn được Appel mô tả trong [Appe67]. Appel đã giới thiệu khái niệm **tính vô hình có định lượng** (*quantitative invisibility*) của một điểm, dùng để chỉ số lượng các đa giác **mặt trước** (*front-facing*) che khuất điểm đó. Một điểm là hiện nếu và chỉ nếu tính vô hình có định lượng của điểm đó là 0. Việc xác định đoạn thẳng hiện đã không còn quan trọng nữa từ khi máy tính trở nên mạnh hơn.

7.1. Loại bỏ mặt quay vào trong

Cách kết xuất đơn giản nhất trong một chương trình dựng hình là hiển thị toàn cảnh theo kiểu khung lưới. Những hiển thị kiểu này rất phù hợp với những trường hợp chỉ cần hiển thị hình một cách bao quát và cần tốc độ kết xuất nhanh. Một cách cực kỳ đơn giản để khiến những hiển thị này trở nên chân thực hơn là loại bỏ các cạnh chung giữa những mặt quay vào trong. Mặt quay vào trong của một đối tượng đặc là một mặt hướng ngược lại với máy quay. Để giải thích điều này và thực hiện phép kiểm tra ta cần tìm vec-tơ pháp tuyến. Mỗi mặt của một khối đặc sẽ tương ứng với một vec-tơ pháp tuyến hướng ra bên ngoài. Mặt khác, lựa chọn vec-tơ pháp tuyến cho một mặt cũng tương đương với việc xác định hướng của mặt đó. Bởi vậy, người ta đưa ra định nghĩa chung của một mặt quay vào trong như sau (xem Hình 7.1):

***Định nghĩa**: một mặt có hướng sẽ được coi là mặt quay vào trong đối với vec-tơ v (thông thường là hướng nhìn của máy quay) nếu góc giữa vec-tơ pháp tuyến n của mặt đó và vec-tơ v nằm trong khoảng từ 0 đến 90 độ. Chính xác hơn, điều này có nghĩa là $n \bullet v \geq 0$. Nếu $n \bullet v \leq 0$ thì mặt đó được coi là mặt trước đối với v.*

Lưu ý rằng việc loại bỏ mặt quay vào trong chỉ là một bước tiền xử lý. Không có gì đảm bảo những mặt còn lại thật sự là mặt hiện. Chúng có thể bị đối tượng khác che khuất hoặc thậm chí bị phần nào đó của cùng một đối tượng đang xét che khuất nếu đối tượng đang xét không lồi.

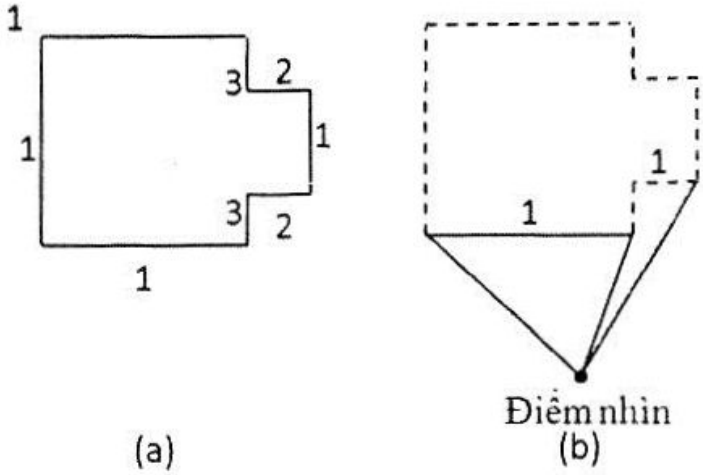


Hình 7.1. Định nghĩa một mặt quay vào trong.

7.2. Thuật toán ưu tiên theo danh sách Schumacker

Thuật toán ưu tiên theo danh sách xuất phát từ [SBGS69]. Đóng góp chính của nó là ý tưởng các mặt của các đối tượng đôi khi có thể được gán cho thứ tự ưu tiên sao cho tính hiện hữu của chúng có thể được tính toán độc lập với điểm nhìn. Hình 7.2 cho thấy một ví dụ của trường hợp này. Trong đó, Hình 7.2a cho thấy một số mặt được đánh số theo độ ưu tiên. Hình 7.2b cho thấy cách sử dụng độ ưu tiên này. Sau khi điểm nhìn đã được xác định, người ta sẽ loại bỏ các mặt quay vào trong (biểu diễn bằng các đường gạch) và sau đó căn cứ vào độ ưu tiên của những mặt còn lại để xác định xem mặt nào là "đứng trước" những mặt nào. Chính xác hơn, nếu một mặt *A* có độ ưu tiên thấp hơn mặt *B* thì có nghĩa là mặt *B*

sẽ không bao giờ che khuất mặt *A*. Trong hình vẽ, ta đi đến kết luận là mặt có độ ưu tiên 2 sẽ không che khuất những mặt có độ ưu tiên 1. Khi có độ ưu tiên, ta có thể áp dụng thuật toán được gọi là **thuật toán của người thợ sơn** (*Painter's algorithm*). Tên của thuật toán này xuất phát từ cách mà người thợ sơn sơn một bức tranh sơn dầu, khi đó một lớp sơn mới được quét lên sẽ phủ lớp sơn cũ.



Hình 7.2. Đánh thứ tự ưu tiên cho các mặt.

Dĩ nhiên, thứ tự ưu tiên mà Schumacker tìm kiếm không phải lúc nào cũng tồn tại. Tuy nhiên, ta có thể chia các đối tượng thành các cụm, trong đó các mặt có thể được gán độ ưu tiên theo cách trên. Các cụm này sau đó cũng sẽ phải có thứ tự ưu tiên.

```
Cho trước: Các mặt của cảnh vật được đưa ra theo
thứ tự từ sau đến trước, có nghĩa là nếu mặt A
nằm sau mặt B hoặc ngang nhau, thì mặt B nằm
trước mặt A và mặt A không bao giờ che khuất mặt
B.

Thực hiện: Vẽ các mặt theo thứ tự từ sau ra
trước.
```

Thuật toán 7.2.1. Thuật toán của người thợ sơn.

7.3. Sắp xếp theo chiều sâu Newell-Newell-Sancha

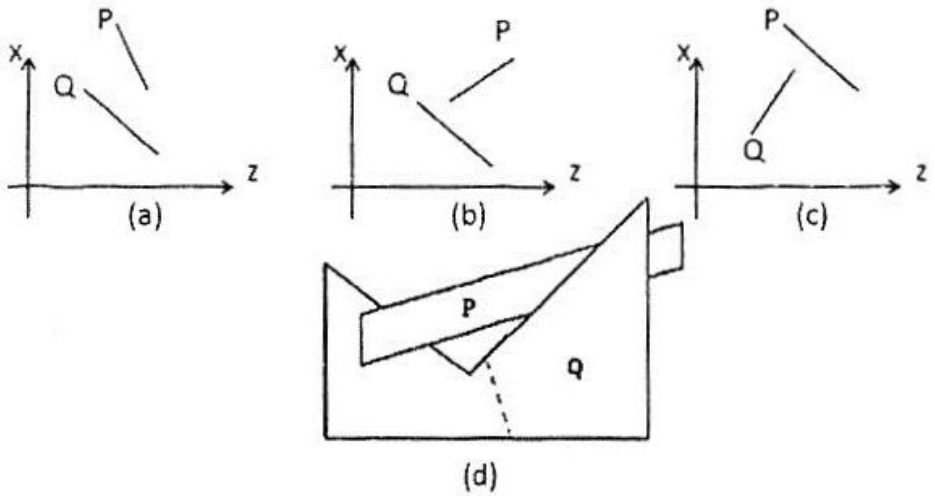
Thuật toán xác định mặt hiện Newell-Newell-Sancha [NeNS72] là thuật toán độ ưu tiên theo danh sách ra đời khá sớm nhằm sắp xếp các đối tượng theo chiều sâu và sau đó sử dụng thuật

toán của người thợ sơn để hiển thị chúng. Điểm khác biệt quan trọng giữa thuật toán này Newell-Newell-Sancha và thuật toán Schumacker là nó tính toán thứ tự chiều sâu và không dựa vào thứ tự ưu tiên như thuật toán Schumacker. Vì lý do này mà thuật toán Newell-Newell-Sancha được đánh giá là linh hoạt hơn. Nó đưa ra nhiều ý tưởng hay trong vấn đề làm cách nào để có được thứ tự chiều sâu.

Thuật toán Newell thực hiện khởi tạo thô thứ tự ban đầu của các mặt đa giác dựa trên giá trị z của đỉnh xa nhất của bề so với điểm nhìn. Tiếp đó, bắt đầu với đa giác P cuối cùng trong danh sách (đa giác nằm ở xa nhất so với điểm nhìn), đa giác tiếp theo Q, thuật toán kiểm tra xem P có thể được vẽ ra ngay thông qua việc xét xem P và Q có tách biệt nhau về độ sâu, nghĩa là:

(Giá trị z nhỏ nhất của một đỉnh thuộc P) > (giá trị z lớn nhất của một đỉnh thuộc Q) ?

Nếu đúng thì P sẽ không bao giờ che khuất bất kỳ một phần nào của Q và ta có thể vẽ P ra ngay (xem Hình 7.3a). Còn nếu không, phải xét tiếp tập các đa giác $\{QS\}$ giao P theo chiều sâu.



Hình 7.3. Một số vị trí tương đối giữa hai mặt.

Mặc dù theo trục z thì các mặt giao nhau nhưng trong thực tế P có thể không che khuất bất kỳ phần nào của các đa giác trong QS. Thuật toán sẽ thực hiện một loạt các phép thử lần lượt theo độ phức tạp. Các phép thử dùng để trả lời các câu hỏi sau:

(1) Có thể phân tách P và tập QS theo x được không?

(2) Có thể phân tách P và tập QS theo y được không?

(3) P có nằm ở phần xa của tập QS hay không (Xem Hình 7.3b)?

(4) Tập QS có nằm ở phần gần của P hay không (Xem Hình 7.3c)?

(5) Hình chiếu của P và tập QS có rời rạc không?

Phép thử (5) rõ ràng là phức tạp nhất và hy vọng là sẽ không phải thực hiện thường xuyên. Nếu câu trả lời cho toàn bộ các phép thử này là "không", thì thuật toán sẽ hoán đổi *P* với một mặt trong tập *QS* và lặp lại các phép thử trên. Thuật toán phải đánh dấu các mặt trong *QS* để tránh gặp phải vòng lặp vô hạn. Cố gắng hoán đổi một phần tử đã được hoán đổi ở lần trước sẽ khiến thuật toán rơi vào trường hợp "giao nhau vòng tròn". Ví dụ minh họa cho trường hợp này được đưa ra trong Hình 7.3d. Lúc này, thuật toán sẽ cắt *Q* tại đường gạch mờ và thay *Q* bằng 2 đa giác mới.

Thuật toán Newell ngoài việc thú vị về mặt lịch sử, nó còn có ý nghĩa ở các bước thử (1)-(4). Chúng rất có ích trong các trường hợp khác.

7.4. Thuật toán BSP

Thuật toán phân vùng không gian nhị phân (*Binary Space Partitioning* - BSP) ([FuKN80] và [FuAG83]) là thuật toán mặt hiện được cải tiến từ thuật toán Schumacker bằng cách tự động chia thành các cụm. Thuật toán BSP nguyên bản gồm 2 bước:

(1) Bước tiền xử lý một lần sẽ chuyển danh sách đa giác đầu vào sang dạng cấu trúc cây nhị phân được gọi là cây BSP.

(2) Một thuật toán duyệt sẽ duyệt qua cây BSP và vẽ các đa giác ra bộ đệm khung theo thứ tự từ sau ra trước.

Ý tưởng chủ đạo ở đây giống như thuật toán của Schumacker chính là sử dụng mặt phẳng phân tách. Mặt phẳng này phải thỏa mãn điều kiện là không có đa giác nào nằm ở nửa không gian chứa

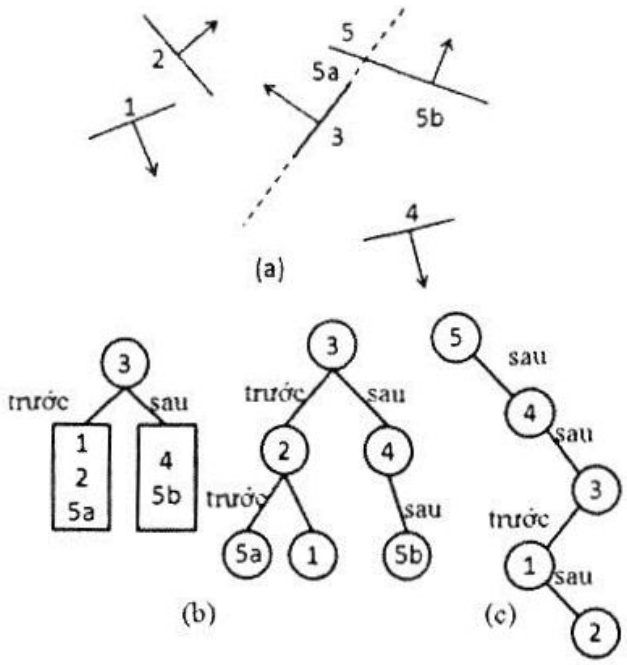
điểm nhìn bị một đa giác nằm ở nửa không gian kia che khuất. Để xây dựng cây BSP, các bước được thực hiện như sau:

(1) Chọn đa giác P bất kỳ từ danh sách hiện tại và đặt đa giác này vào gốc.

(2) Các đa giác còn lại trong danh sách sẽ được kiểm tra xem chúng thuộc nửa không gian chia bởi mặt phẳng chứa P. Nếu chúng nằm cùng phía với điểm nhìn thì ta gán chúng vào cành bên trái (hay cành "mặt trước"), ngược lại chúng sẽ được gắn vào cành phải (hay cành "mặt sau"). Nếu một đa giác giao với mặt phẳng chứa P thì chia nó ra làm đôi bởi mặt phẳng này và gán mỗi nửa đa giác vào cành con tương ứng.

(3) Lặp lại các bước trên đối với 2 cành con trái và phải.

Xét ví dụ trong Hình 7.4. Có 5 đa giác được đánh số từ 1 đến 5 trong Hình 7.4a. Giả sử các mũi tên chỉ ra về phía có điểm nhìn (Trong ví dụ trong Hình 7.4, các hướng được chọn chỉ là ví dụ, không tương ứng với trường hợp thực tế nào). Ta giả thiết đa giác số 3 được chọn đầu tiên, sau đó là 2 và cuối cùng là 4. Các bước xây dựng cây BSP được chỉ ra trong Hình 7.4b. Lưu ý rằng việc lựa chọn các đa giác có thể có ảnh hưởng lớn đến kết quả đầu ra. Hình 7.4c cho ta thấy cây BSP mà ta xây dựng được nếu ta chọn các đa giác theo thứ tự lần lượt là 5, 4, 3, 1 và 2.



Hình 7.4. Một ví dụ cây BSP.

Khi đã dựng được cây BSP, thì không khó để tạo khung nhìn bằng cách duyệt cây theo thứ tự. Tại mỗi nút trên cây ta xác định xem liệu điểm mắt nằm trước hay sau gốc hiện tại của cây. Duyệt phía bên kia của cây, xuất đa giác gốc sang bộ đệm khung và sau đó duyệt nốt phía còn lại.

Thuật toán 7.4.1 là một minh họa rõ ràng hơn nữa về cách xây dựng cây BSP và cách duyệt cây. Lúc đầu, người ta lo ngại rằng cây BSP sẽ lớn hơn đáng kể so với danh sách đa giác đưa vào nhưng điều đó đã không xảy ra trong thực tế. Để giảm bớt số đa giác bị chia, người ta áp dụng quy tắc tham lam sau: chọn đa giác sẽ trở thành gốc của cây con tiếp theo sao cho chỉ phải cắt bỏ lượng ít nhất các đa giác khác. Người ta phát hiện ra rằng không nhất thiết phải tìm kiếm qua toàn bộ các đa giác còn lại để tìm đa giác ở gốc mà chỉ cần chọn đa giác tốt nhất trong một số đa giác được chọn ra một cách ngẫu nhiên (khoảng 5 đa giác là cho kết quả tốt).

```
BSPtree BuildBSPTree(PolygonList plist)
{
      PolygonList frontList, backList;
      Polygon root, poly, frontPart, backPart;

      if (IsEmpty(plist)) return (EmptyBSPTree);

      frontList = nil;
      backList = nil;

      root = SelectAndRemoveOne (plist);

      for (poly in plist)
      {
            if   (InFrontOf   (poly,root))   Insert
      (poly,frontList);
            else if InBackOf (poly,root) Insert
      (poly,backList);
            else
            {
```

```
                SplitPolygon
            (poly,root,frontPart,backPart);
                Insert (frontPart,frontList);
                Insert (backPart,backList);
            }

            return   (CreateBSPTree   (BuildBSPTree
        (frontList),root,BuildBSPTree (backList)));
        }
}

// Thủ tục Display được giả thiết là sẽ thực hiện
tất cả các bước biến đổi, tạo bóng, v.v.

void TraverseBSPTree (BSPTree T)
{
        if (IsEmpty (T)) then return;
        if (InFrontOf (eye,rootPolygon (T)))
        {
            TraverseBSPTree (backSubtree (T));
            Display (rootPolygon (T));
            TraverseBSPTree (frontSubtree (T));
        }
        else
        {
            TraverseBSPTree (frontSubtree (T));
            TraverseBSPTree (backSubtree (T));
        }
}
```

Thuật toán 7.4.1. Thuật toán cây BSP.

Ta ta vừa mô tả thuật toán BSP nguyên gốc. Nhiều biến thể của thuật toán này cũng được phát triển sau đó. Ví dụ, Kaplan [Kapl85] chọn mặt phẳng phân tách song song với các mặt phẳng tọa độ của hệ tọa độ tham chiếu. Đôi khi thuật toán này cũng được gọi là thuật toán BSP trực giao (orthogonal BSP algorithm). Tính chất này của mặt phẳng phân tách này có thể làm đơn giản hóa các tính toán.

Tổng kết lại, thuật toán BSP là một thuật toán tốt khi mô hình ít thay đổi và chỉ có điểm nhìn thay đổi. Các ví dụ trong trường hợp này là các trình giả lập lái máy bay, mô tả cảnh các kiến trúc sư đi dạo quanh một ngôi nhà, hay các nhà hóa sinh quan sát những phân tử phức tạp, v.v. Chin [Chin95] đã đưa ra một cách thức cài đặt và sử dụng cây BSP rất hiệu quả.

7.5. Phép chia nhỏ từng vùng Warnock và Weiler - Atherton

Thuật toán xác định mặt hiện Warnock [Warn69] là một thuật toán chính xác theo ảnh trong đó người ta cố gắng tìm những mảnh nhỏ hình chữ nhật (gọi là cửa sổ) có cùng màu sắc/cường độ trên màn hình (khu vực liên kết). Thuật toán 7.5.1 cho thấy ý tưởng chính chính của thuật toán này. Đa giác được nhắc đến trong thuật toán là các đa giác được chiếu (xem Hình 7.5). Cốt lõi của thuật toán này là khả năng thực hiện các bước thử nghiệm sau trên đa giác P bất kỳ:

Phép thử 1: P có tách biệt với cửa sổ không?
Phép thử 2: P có bao cửa sổ không?
Phép thử 3: P có giao một phần với cửa sổ không?
Phép thử 4: P có nằm trong cửa sổ không?
Phép thử 5: P có nằm trước các đa giác khác không?

Khởi tạo danh sách các cửa sổ *L* gồm các cửa sổ đơn chính là toàn bộ cửa sổ màn hình;
Với mỗi cửa sổ *W* trong danh sách hiện thời *L* tìm một cửa sổ thỏa mãn một trong 3 trường hợp sau:

(1) Tất cả các đa giác đều tách biệt với *W*. Trong trường hợp này vẽ *W* với màu nền.

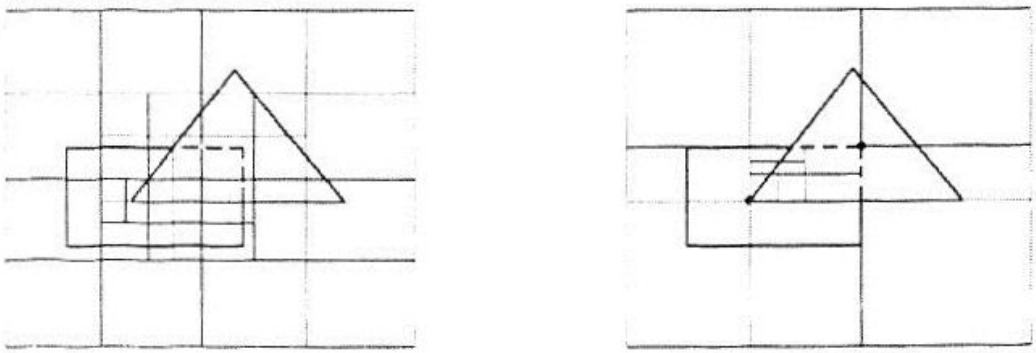
(2) Chỉ có một đa giác *P* giao với *W*. Trong trường hợp này vẽ phần giao của *P* và *W* với màu của *P* và phần còn lại của *W* với màu nền. Trong thực tế, trường hợp này lại được chia thành 3 trường hợp con khác là: *P* nằm trong *W*, *P* bao phủ *W*, và các trường hợp *P* và *W* giao nhau khác.

(3) Tìm ra được một đa giác bao quanh *W* và đa giác đó nằm trước tất cả các đa giác giao với

cửa sổ. Trong trường hợp này vẽ cửa sổ với màu của đa giác.

Trong trường hợp khác, ta chia nhỏ cửa sổ *W* thành 4 cửa sổ con đều nhau và đưa chúng vào danh sách *L*, lặp lại quá trình xử lý cho đến khi ta hạ kích thước cửa sổ xuống thành 1 điểm. Lúc này ta có thể kiểm tra ngay lập tức đa giác bất kỳ xem nó có nằm trước tất cả các đa giác còn lại tại điểm này hay không.

Thuật toán 7.5.1. Ý tưởng của thuật toán Warnock.



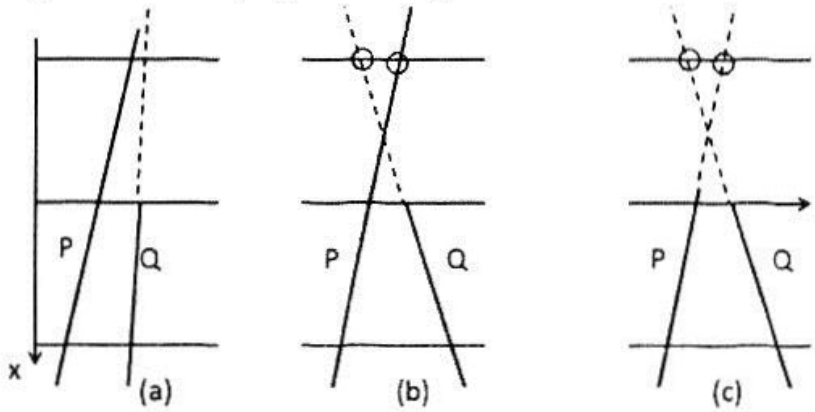
Hình 7.5. Ví dụ về phép chia nhỏ từng vùng.

Để có thể kiểm tra nhanh sự tách biệt giữa cửa sổ và đa giác, người ta thường sử dụng hộp bao. Một cách kiểm tra xem cửa sổ có nằm trong một đa giác nào đó không là thay tọa độ các đỉnh của cửa sổ trong công thức bằng các cạnh của đa giác. Nếu các phép thử trên sai, thì ta cần kiểm tra tiếp liệu đường biên của các đa giác có giao với các cửa sổ không bằng cách kiểm tra từng cạnh của đa giác với từng cạnh của cửa sổ. Nếu các đường biên này tách biệt nhau, ta vẫn phải phân hai trường hợp: các vùng tách biệt nhau hay chứa nhau. Một cách khác là sử dụng phép thử chẵn lẻ và đếm số lần một tia ngang xuất phát từ một đỉnh của cửa sổ giao với đa giác đang xét. Nếu số giao điểm là chẵn, thì có 2 hình tách biệt nhau, nếu không thì đa giác bao cửa sổ. Một cách tiếp cận khác là sử dụng tham số đếm góc.

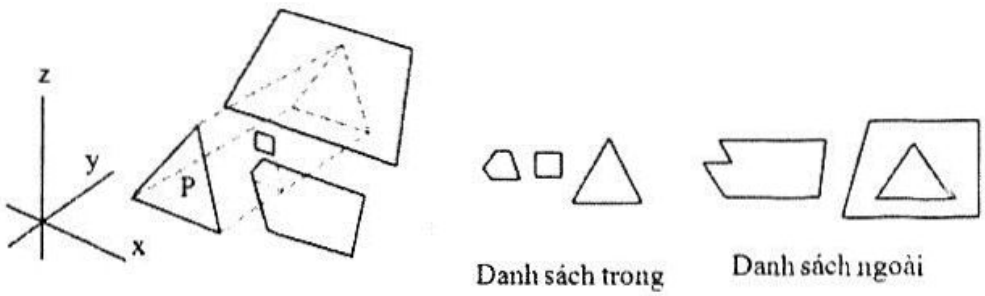
Các phép thử 1-4 ở trên xử lý vấn đề mặt phẳng nhìn. Phép thử thứ 5 liên quan đến những tính toán về độ sâu. Như đã nói ở trên, ta giả thiết về phép chiếu trực giao với máy quay ở vị trí vô cùng và như thế thì kiểm tra xem liệu một điểm có nằm trước các điểm khác hay không chính là kiểm tra xem giá trị z của điểm đó có nhỏ hơn

giá trị z của các điểm khác không. Giả sử hai đa giác P và Q giao với cửa sổ. Ta sẽ thử xem liệu P có nằm trước Q hay không. Theo thuật toán Warnock thì phép thử này chỉ cần thiết trong trường hợp P là đa giác bao ngoài nhưng trong thực tế thì việc nó có bao ngoài hay không không quan trọng. Phép thử được thực hiện như sau: nếu độ sâu của mặt phẳng chứa P nhỏ hơn chiều sâu của mặt phẳng chứa Q tại bốn góc cửa sổ thì P nằm trước Q (Hình 7.6a). Đây là điều kiện đủ nhưng không nhất thiết phải là điều kiện cần để P nằm trước Q như trong Hình 7.6b và Hình 7.6c. Warnock chia nhỏ cửa sổ nếu phép thử thất bại.

Có rất nhiều biến thể khác nhau của thuật toán Warnock. Các cửa sổ cũng không nhất thiết phải là hình chữ nhật. Vấn đề xảy ra với cửa sổ hình chữ nhật là ở sự bất cân xứng của hầu hết các đa giác, thuật toán phải chạy đệ quy xuống rất sâu tận mức điểm. Thuật toán Weiler-Atherton [WeiA77] sử dụng cửa sổ con để tạo sự cân xứng cho hình dạng của đa giác.



Hình 7.6. Ví dụ về một mặt nằm trước một mặt khác.



Hình 7.7 Chia nhỏ theo khu vực kiểu Weiler-Atherton.

Xem Hình 7.7 ta thấy một danh sách các đa giác đã bị cắt bỏ dựa trên đa giác P. Các mảnh cuối cùng được chia ra thành 2 danh

sách, một danh sách gồm các mảnh nằm trong P và danh sách kia gồm các mảnh không nằm trong P. Việc thực hiện sẽ phức tạp hơn và tác giả phải xây dựng thuật toán cắt mới, tất cả những vấn đề này được đề cập chủ yếu trong mục 3.3.2, nhưng bù lại ta sẽ thu được hiệu quả tốt hơn.

Cả thuật toán Warnock và Weiler-Atherton đều không theo hướng quét đường thẳng nhưng có liên quan đến khi vẽ khung cảnh. Độ phức tạp của 2 thuật toán là xấp xỉ với độ phức tạp của việc hiển thị kết quả sau cùng (chứ không phải xấp xỉ với độ phức tạp của khung cảnh).

7.6. Các thuật toán bộ đệm Z

Các thuật toán bộ đệm Z ghi lại thông tin về độ sâu hiện thời của mỗi điểm. Một bộ đệm Z chính là một mảng 2 chiều chứa các số thực, có kích thước giống như bộ đệm khung. Các số thực thể hiện thông tin về chiều sâu hiện thời. Thuật toán bộ đệm Z sẽ quét rồi chuyển toàn bộ các mặt vào cả bộ đệm khung và bộ đệm Z. Ý tưởng chính của các thuật toán dạng này rất đơn giản, được thể hiện trong Thuật toán 7.6.1.

```
/*Gọi Depth(p) là độ sâu của điểm p, chính là tọa
độ z của p trong hệ tọa độ mắt hay hệ tọa độ máy
quay.
Mảng DEPTH dưới đây chứa các giá trị z của các
điểm gần mắt nhất.*/

color FRAMEBUF[XMAX - XMIN + 1][YMAX - YMIN + 1];
double DEPTH[XMAX - XMIN][YMAX - YMIN + 1];
{
      Khởi tạo FRAMEBUF với màu trùng với màu nền;
      Khởi tạo DEPTH bằng ∞;
      for (tất cả các bề mặt F trong cảnh vật)
          for (mỗi điểm p thuộc F)
          if (p chiếu sang mảng
            FRAMEBUF[i - XMIN][j - YMIN] với i, j
nào đó)
                  if (Depth (p) < DEPTH[i - XMIN][j
- YMIN]){
                        FRAMEBUF[i - XMIN][j - YMIN]
                              = màu của   F tại p;
```

```
                        DEPTH[i - XMIN][j - YMIN] =
Depth (p);
                    }

}
```

Thuật toán 7.6.1. Thuật toán bộ đệm Z.

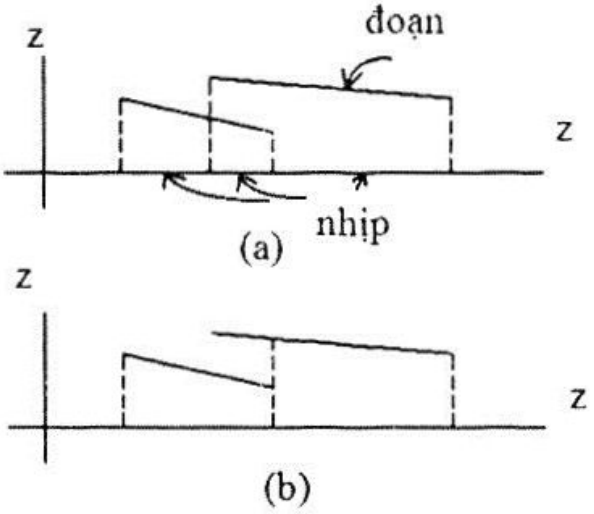
Các biến của thuật toán bộ đệm Z kết hợp chặt chẽ với kỹthuật khử răng cưa. Một thuật toán dòng quét nữa được mô tả trong [Catm78] cũng có chi phí tính toán rất cao.

Bộ đệm Z tốn rất nhiều bộ nhớ. Để cải tiến, ta kết hợp vớithuật toán dòng quét, do đó chỉ cần một mảng tương ứng với một dòng quét. Điều này dẫn tới việc ta phải sửa chút ít thuật toán bộ đệm Z như trong Thuật toán 7.6.2 (xem [Roge98]). Các thuật toán bộ đệm Z đều sử dụng khái niệm "**đoạn**" (*segment*) và "**nhịp**" (*span*) trong dòng quét (nơi một mặt giao cắt với dòng quét này) và đặt n câu hỏi "Đoạn nào là đoạn hiện?" Ví dụ trong Hình 7.8, với mặt phẳng x-z, đường gạch đứt chia các đường quét thành các nhịp. Trong mỗi nhịp, thuật toán sẽ xác định những đoạn có thể nhìn được bằng cách kiểm tra độ sâu của chúng theo công thức mặt phẳng. Có nhiều cách chia ra các nhịp vì yêu cầu duy nhất là chia như thế nào để các đoạn có thể được sắp xếp theo độ sâu một cách rõ ràng. Bởi vậy, một vấn đề đặt ra là chọn nhịp ra sao để có kết quả tốt nhất. Ví dụ, Hình 7.8(b) cho ta thấy cách lựa chọn nhịp tốt hơn cách lựa chọn ở Hình 7.8(a).

```
/*Giả sử cảnh vật được chuyển đổi vào hệ tọa độ mà
trong đó mỗi điểm trên màn hình tương ứng với các
điểm tọa độ nguyên của hình chữ nhật [XMIN,
XMAX]X[YMIN,YMAX] và ta đang sử dụng phép chiếu
trực giao (máy quay ở vị trí -∞ trên trục z). Hàm
Depth(p) trả lại tọa độ z của điểm p và ixp là tọa
độ x của p được làm tròn đến số nguyên gần nhất */

color COLOR[XMAX - XMIN + 1];
/* mảng lưu màu của các điểm trên dòng quét hiện
tại */
double DEPTH[XMAX - XMIN + 1];
```

```
/* mảng lưu giá trị z của điểm nằm gần mắt nhất */

for (iy = YMIN; iy <= YMAX; iy++){
     /* với mỗi dòng quét của màn hình */
      Khởi tạo mảng COLOR với màu nền;
      Khởi tạo mảng DEPTH với các giá trị ∞;
      for (với từng mặt được chiếu F) {
           if (F giao với mặt phẳng y == iy){
                Gán L là đoạn giao này;
                for (với mỗi p thuộc L)
                     if (Depth (p) < DEPTH [ixp -
XMIN]){
                          COLOR  [ixp  -  XMIN]  =
màu của S;
                          DEPTH  [ixp  -  XMIN]  =
Depth (p);
                     }
           }
      Hiển thị COLOR;
}
```

Thuật toán 7.6.2. Thuật toán bộ đệm Z dòng quét.



Hình 7.8. Các nhịp (span) trong thuật toán bộ đệm Z.

7.7. Thuật toán dòng quét Watkins

Thuật toán Watkins [Watk70] là một thuật toán dòng quét trong không gian ảnh đặc biệt hiệu quả. Ý tưởng của thuật toán này là để đầu mút phải của nhịp hiện tại "biến đổi" và giữ cố định đầu

mút trái. Thuật toán bắt đầu với đầu mút phải xa nhất. Khi lấy ra các đoạn mới từ danh sách đã sắp xếp theo x, đầu mút phải sẽ được điều chỉnh dần về bên trái cho đến khi thu được một nhịp đơn giản đủ để tính được đoạn nào là hiện.

Thuật toán 7.7.1 cho ta một minh họa cho kiểu thuật toán Watkins. Một điều cần lưu ý ở đây là ta giả thiết các đa giác này đã được cắt xén theo khung hình. Để giúp hiểu hơn thuật toán thì ta sẽ tìm hiểu một ví dụ. Xét khung cảnh được cho trong Hình 7.9(a). Hình 7.9(b) và Hình 7.9(c) cho thấy hình chiếu của nó trên mặt phẳng x-z và x-y. Giả sử ta đang ở dòng quét y như trong Hình 7.9(c). Các điểm quan trọng dọc theo trục x là 0, a, b, c và d. Bảng 7.1 minh họa việc quét x được thực hiện như thế nào. Trong bảng đó, các giá trị của các biến ở các vị trí đánh dấu "LA" hoặc "LB" trong thủ tục PrecessActiveEdgeList được đưa ra. Các đối tượng trong Hình 7.9(a) không cắt nhau. Hình 7.10 chỉ ra một trường hợp một đa giác đan xen vào một đa giác khác. Trong trường hợp này ta phải tìm phần giao và dịch phần mút phải của nhịp sang bên trái, lưu lại các giá trị hiện thời cho đến khi ta tìm được một đoạn không có phần giao nào. Trong hình vẽ ta thấy có một phần giao có tọa độ là x_I trong đoạn [b, c]. Ta cần kiểm tra tất cả các đa giác đang xét với tất cả các đa giác đang xét khác khi tìm các phần giao. Chúng ta có thể biết hai đa giác có cắt nhau không bằng cách xét giá trị chiều sâu z của chúng tại các đầu mút của nhịp. Gọi zl_1 và zr_1 là các giá trị của đa giác thứ nhất và zl_2, zr_2 là các giá trị của đa giác thứ hai. Các đa giác này sẽ giao nhau nếu:

$$((zl_1 < zl_2) \textbf{ and } (zr_1 > zr_2)) \textbf{ or } ((zl_1 > zl_2) \textbf{ and } (zr_1 < zr_2)).$$

Nếu ta tìm thấy một phần giao, một phần của thủ tục LastVisiblePolygonColor sẽ được thực hiện. Việc quét x mới sẽ chạy qua các lệnh đánh dấu "LC" và "LD" như ta thấy trong Bảng 7.2. Đầu tiên ta lưu *c* vào ngăn xếp và xử lý đoạn [b, x_I]. Sau đó ta gán spanLeft bằng x_I, lấy c ra khỏi ngăn xếp và xử lý đoạn [x_I, c]. Tiếp tục lặp lại những thao tác như trước.

Chú ý là ta luôn phải kiểm tra xem các đoạn có giao với điểm mút của nhịp hay không. Nếu trường hợp này xảy ra, ta sẽ dùng giá

tr x của điểm mút còn lại để thực hiện phép kiểm tra chiều sâu để xem đa giác nào gần hơn.

```
Giả thiết khung nhìn là [XMIN,XMAX] x [YMIN,YMAX].
/*Bản ghi dữ liệu ta sẽ cần dùng cho từng đa giác
trong không gian sự vật*/

struct polydata{
     int y; /* dòng quét đầu tiên bị cắt bởi đa
giác */
     edgedata list edges;
     /* mỗi cạnh của đa giác có một bảng ghi dữ
liệu */
     double a, b, c, d;
     /* ax + by + cz + d = 0
     là phương trình mặt phẳng của đa giác */
     color hue;
     bool active;
     /* sẽ được cập nhật khi chạy thuật toán */
};

/* bản ghi lưu điểm mút của nhịp ngang */
struct edgedata{
     polydata* polyP;
 /*con trỏ lưu giữ dữ liệu của đa giác chứa cạnh
nào đó*/
     double x, /* tọa độ của điểm là giao giữa cạnh
đang xét và dòng quét hiện thời */
     dx; /*độ nhảy của x từ dòng quét này sang dòng
quét tiếp theo */
     int dy;
     /*số đường quét còn lại sẽ được cắt bởi cạnh*/
}

/* các biến toàn cục: */
polydata list activePolys;
/* danh sách các đa giác active hiện thời */
edgedata list activeEdges;
/* danh sách các cạnh active hiện thời */
polydata list array buckets[YMAX - YMIN + 1];
/* buckets[y] lưu giữ tất cả các đa giác nằm ở đầu
dòng quét y*/
double spanLeft, spanRight;
/*used by void ProcessActiveEdgeList */
```

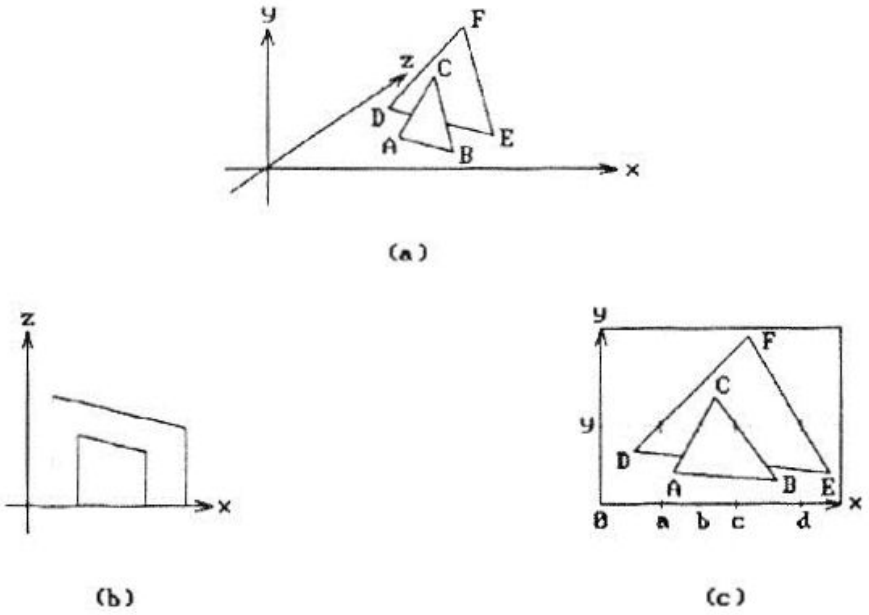
```
void WatkinsAlgorithm (){
      int y;
      InitializeData ();
      for (y = YMIN; y <= YMAX; y++){
            Thêm các đa giác trong mảng buckets
            [y - YMIN]vào danh sách đa giác active
            activePolys;
            Quét các cạnh của đa giác trong
            activePolys và bổ sung thêm những cạnh
            này tại mỗi y mới vào danh sách các cạnh
            active activeEdges;
            Sắp xếp lại các cạnh trong activeEdges
            mỗi khi x tăng;
            /*Việc sắp xếp lại là cần thiết vì khi
            ta cập nhật danh sách thì thứ tự cũ sẽ
            bị làm hỏng nếu các cạnh lại chen lộn
            xộn vào nhau*/
            ProcessActiveEdgeList ();
            UpdateActiveEdgeList ();
            UpdateActivePolygonList ();
      }
}

void InitializeData (){
      int i;
      activePolys =  null;
      activeEdges =  null;
      { Initialize all buckets to nil }
      for (i = YMIN; i <= YMAX; i++)
          buckets[i - YMIN] = null;
      for (với mỗi đa giác P trong không gian sự
            vật){
            Tạo một bản ghi dữ liệu đa giác mới cho P
                (trường acitve đặt là faslse;
            Thêm pData vào buckets[pData.y];
      }
}

void ProcessActiveEdgeList (){
      color spanColor;
      int polyCount;
      edgedata E;
      polydata P;
```

```
      spanLeft = XMIN;
      polyCount = 0;
      for (E in activeEdges){
            spanRight = E.x;
{LA}  switch (polyCount){
                  case 0 :
                        spanColor = backgroundColor;
break;
                  case 1 :
                        spanColor      =      ColorOf
(OnlyMemberOf
                     (activePolys)); break;
                  default :
                        spanColor                    =
LastVisiblePolygonColor;
                     break;
            }
            P = PolydataOf (E);
            ToggleActive (P);
           /*if active field true, set to false and
vice
           versa */
            if (IsActive (P))
                  polyCount = polyCount + 1
            else
                  polyCount = polyCount - 1; {LB}

            Display
([spanLeft,spanRight],spanColor);
            spanLeft = spanRight;
      }
      if (spanLeft < XMAX)
            Display
([spanLeft,XMAX],backgroundColor);
}

void CheckForSegIntersections (bool& intersected,
double& xint){
      if (2 đoạn trong danh sách activeEdges giao
nhau tại tọa độ x = xint với spanLeft < xint <
spanRight)
            intersected = true;
      else{
            intersected = false;
            if (2 đoạn trong activeEdges wo segments
```

```
in
                activeEdges  gặp  nhau  tại  vị  trí
spanLeft)
                    xint =  spanRight
               else
                    xint =  spanLeft;
          }
}

polydata  MinActiveZvaluePolygon  (double  x,double
y){
/* Quét qua danh sách các đa giác active và trả lại
giá trị z nhỏ nhất tại (x, y). Các giá trị z được
xác định bằng công thức z = -(a*x + b*y + d)/c
trong đó a,b,c, và d là hệ số của mặt phẳng đa
giác.*/

color LastVisiblePolygonColor ()
/* Kiểm tra lại các giao cắt, là vị trí cần tến
trong nhiều vấn đề */
{
     double stack spanStack;
     bool intersected;
     double xint;
     Gán cho stack của nhịp spanStack giá trị rỗng;
     while (true){
          CheckForSegIntersections
(intersected,xint);
          if (intersected){
               /* Có 1 giao cắt. Đẩy giá trị
                spanRight hiện tại vào stack, của
                nhịp ra, xử lý nửa trái của nhịp
                [spanLeft, xint] */
               Push (spanright,spanStack);
               spanRight =  xint;
{LC}  } else{
               Segcol                              =
ColorOf(MinActiveZvaluePolygon (xint,y));
               if (Empty (spanStack))
                    return (segcol);
               Display                    ([spanLeft,
spanRight],segcol);
{LD}       spanLeft =  spanRight;
               spanRight =  Pop (spanStack);
          }
```

```
    }
} /* LastVisiblePolygonColor */
```

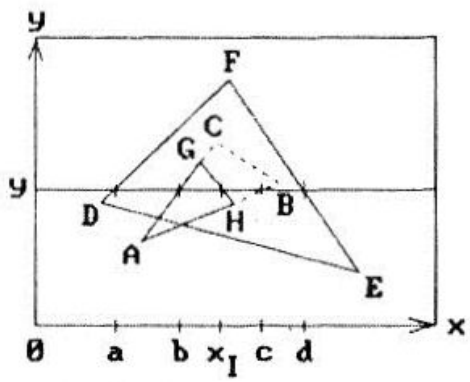
Thuật toán 7.7.1. Thuật toán bề mặt hiện Watkins.



Hình 7.9. Một ví dụ về thuật toán Watkins.

Label	polyCount	spanLeft	spanRight	ABC.active	DEF.active
LA	0	0	a	F	F
LB	1	0	a	F	T
LA	1	a	b	F	T
LB	2	a	b	T	T
LA	2	b	c	T	T
LB	1	b	c	F	T
LA	1	c	d	F	T
LB	0	c	d	F	F

Bảng 7.1. Dữ liệu quét x tại dòng quét y cho Hình 7.9(c).



Hình 7.10. Thuật toán Watkins với các đối tượng xen vào nhau.

Label	polyCount	spanLeft	spanRight	ABC.active	DEF.active	spanStack
LA	0	0	a	F	F	ni
LB	1	0	a	F	T	ni
LA	1	a	b	F	T	ni
LB	2	a	b	T	T	ni
LA	2	b	c	T	T	ni
LC	2	b	x_1	T	T	-
LD	2	x_1	c	T	T	ni
LB	1	x_1	c	F	T	ni
LA	1	c	d	F	T	ni
LB	0	c	d	F	F	ni

Bảng 7.2. Dữ liệu quét x tại dòng quét y cho Hình 7.10.

7.8. Đánh giá

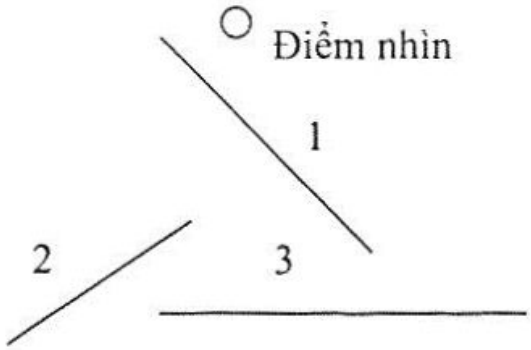
Các thuật toán xác định mặt hiện cần được đánh giá trong ổng thể toàn bộ thuật toán kết xuất. Nói chung, không có khái niệm "tốt nhất" cho thuật toán xác định mặt hiện. Sự khác nhau giữa các dòng thuật toán xuất phát từ những yêu cầu khác nhau: khung ảnh khác nhau và độ phức tạp dữ liệu xuất khác nhau. Việc lựa chọn thuật toán xác định mặt hiện còn phụ thuộc vào việc thuật toái tạo bóng nào được sử dụng. Ví dụ, mặc dù Watkins là một thuật oán hay nhưng người ta không thể chọn nó nếu muốn sử dụng mô ình ánh toàn cục. Nói chung các thuật toán không gian ảnh có vẻ phổ biến hơn cả. Trong thực tế, các thuật toán bộ đệm Z, đơn giải và linh hoạt, là chuẩn phổ biến trong các hệ thống đồ họa chất lượng cao vì giá cả của bộ nhớ không còn phải là vấn đề nữa.

Việc sắp xếp theo chiều sâu và các thuật toán quét dòng sẽ iệu quả nếu số lượng mặt là nhỏ (khoảng vài nghìn). Các thuật oán quét dòng tận dụng sự liên kết giữa các dòng liên tiếp và sự liê kết giữa các nhịp trong cùng một dòng quét. Điều này sẽ giúp chuyển từ bài toán 3 chiều sang bài toán 2 chiều. Ta chỉ ngại rằng cá đa giác vô cùng bé sẽ có thể nằm giữa các dòng quét. Thông thường, việc sắp xếp theo chiều sâu sẽ có hiệu quả nếu khung cảnh trải heo chiều sâu (chồng chéo lên nhau theo chiều z) và thuật toán òng quét hay thuật toán chia nhỏ từng mảnh sẽ có hiệu quả nếu các ặt phân tách nhau rõ ràng theo chiều dọc hay chiều ngang. Với nhưng khung cảnh có số lượng đối tượng lớn thì phương thức bộ đệm sẽ có hiệu quả vì độ phức tạp của chúng không phụ thuộc vào số lượng mặt. Việc sắp xếp là một phần cực kỳ quan trọng của nhưng

thuật toán mà ta đã bàn đến. Thật vậy, sự khác nhau giữa các thuật toán thực sự là do cách chúng tiến hành sắp xếp.

Câu hỏi và bài tập

1. Hãy tạo cây BSP cho trường hợp trong hình sau và sau đó đưa ra thứ tự vẽ



2. Hãy nêu độ phức tạp theo điểm của thuật toán Warnock.
3. **Bài tập lập trình:** Chương trình vẽ - hãy mở rộng chương trình đã phát triển trong bài tập lập trình ở chương trước để có thêm chức năng chỉ ra những mặt tam giác quay vào trong.

Chương 8
ĐƯỜNG CONG VÀ BỀ MẶT

8.1. Giới thiệu

Các chương trước của cuốn sách đã mô tả những ý tưởng cơ bản và các thuật toán thường hay sử dụng để kết xuất các đối tượng hình học, với các chủ đề chính là nền tảng toán học trong luồng xử lý đồ họa, cắt xén, vẽ các đoạn thẳng rời rạc, xác định mặt hiện và tạo bóng. Tóm lại, chúng ta đã nắm vững những kiến thức cần thiết để kết xuất bất cứ khối đa diện tuyến tính nào. Tuy nhiên, các khối đa diện chưa đủ để mô tả hết các đối tượng. Đã đến lúc nói đến các đối tượng cong, một thành phần rất quan trọng trong thiết kế đồ họa. Đây là một lĩnh vực rất rộng và có nhiều vấn đề để nói đến. Tuy nhiên chúng ta chỉ đề cập đến một số nội dung nhất định. Với hi vọng chắt lọc được những kiến thức cốt lõi, chương này mong muốn người đọc thu được những điều sau:

(1) khả năng mô tả những đường cong và bề mặt mà ta thường gặp trong thiết kế đồ họa máy tính,

(2) hiểu biết căn bản về sự quan trọng của những đối tượng cong này,

(3) những thuật toán hiệu quả để tính toán được một số thuộc tính quan trọng nhất của đối tượng,

(4) phương thức xây dựng đối tượng dễ dàng hơn.

Chúng ta bắt đầu với một số nhận xét. Câu hỏi đầu tiên đặt ra là làm thế nào để biểu diễn các đối tượng cong. Một hình đa giác có thể được biểu diễn dưới dạng một dãy các điểm. Tuy nhiên cách này không phù hợp cho các đối tượng cong, dù rằng kiểu gì thì ta cũng phải mô tả chúng một cách hữu hạn trên máy tính. Hai cách

biểu diễn chuẩn là rõ ràng bằng tham số hóa hoặc ẩn qua các phương trình. Biểu diễn ẩn qua các phương trình thường không tiện lợi, ngoại trừ những đường cong và bề mặt nón. Chúng ta sẽ tập trung vào các đường cong và bề mặt tham số.

Cách chúng ta xác định các đường cong và bề mặt phụ thuộc vào mục đích của chúng ta: để định nghĩa, để dễ cài đặt hay để người dùng sử dụng dễ dàng. Thường thì người dùng là yếu tố quyết định chính. Người dùng muốn tạo ra và thao tác với các đối tượng một cách đơn giản nhất. Ví dụ, người ta muốn định nghĩa một đoạn cong lập phương bằng cách đơn giản là xác định một số điểm điều khiển hình dạng, bằng một vài vectơ tiếp tuyến hoặc bằng một vài điều kiện về độ cong của đường. Nhiện vụ của các nhà toán học là đưa ra cách để xác định đường cong này một cách khả thi và hiệu quả.

Khi ta xét xem các đường cong và các bề mặt được sử dụng như thế nào trong mô hình hóa, có hai cách tiếp cận: cách thứ nhất mô hình hóa đối tượng một cách rất chính xác với một sai số cho phép nhất định, và cách thứ hai mô hình theo kiểu phác thảo gần đúng. Ví dụ, trong thiết kế cánh máy bay hoặc cánh quạt tuabin người ta sẽ phải phân tích để mô hình chúng một cách trung thực nhất. Mặt khác, khi thiết kế thân xe ô tô, người ta quan tâm nhiều hơn cảm quan trực giác và tính thẩm mỹ hơn. Lúc này sai số không phải là vấn đề cần tập trung chính.

Nhiều định nghĩa về đường cong và bề mặt xuất phát từ bài toán khớp dữ liệu (data fitting), hay theo một khía cạnh nào đó, là trường hợp đặc biệt của bài toán:

Bài toán xấp xỉ tổng quát: Cho một tập cố định các hàm $\varphi_1, \varphi_2, ..., \varphi_k$, tìm các hệ số c_i sao cho:

$$g(x) = \sum_{i=1}^{k} c_i \varphi_i(x) \qquad (8.1)$$

là một phép tính xấp xỉ đối với một hàm f(x) nào đó. Hàm φ_i thường được gọi là các *hàm cơ sở* (*basic function*).

Ví dụ, cho $f(x)$ là hàm giá trị thực với các biến thực, hàm $\varphi_i(x)$ là đơn thức x^i, thì ta cần tìm đa thức $g(x)$ xấp xỉ tốt nhất $f(x)$. Do hàm g được xác định theo Công thức 8.1 phụ thuộc vào các giá trị c_i, chúng ta sẽ dùng ký hiệu $g(x_j, c_1, c_2, ..., c_k)$ để chỉ rõ sự phụ thuộc này. Hàm g là một phép xấp xỉ "tốt" với các tính chất sau:

(1) Hàm g rất gần f theo một độ đo nào đó;

(2) Các hệ số c_i là duy nhất.

Thường thì chúng ta chỉ biết giá trị của hàm f(x) tại một số hữu hạn các điểm $x_1, x_2, \ldots x_s$. Như vậy, việc tốt nhất chúng ta có thể làm được là tối thiểu sai số tại các điểm x_i. Vì khoảng cách liên quan đến căn bậc hai nên chúng ta có thể đơn giản hóa mà không làm thay đổi vấn đề tối thiểu hóa bằng cách sử dụng bình phương khoảng cách.

Định nghĩa. Hàm $g(x_j, c_1, c_2, ..., c_k)$ mà giảm thiểu

$$E(c_1, c_2, ..., c_k) = \sum_{j=1}^{s} \left(f(x_j) - g(x_j, c_1, c_2, ..., c_k) \right)^2$$

được gọi là xấp xỉ bình phương tối thiểu (least squares approximation) của hàm f(x).

Bài toán này có thể được giải quyết bằng việc đặt các đạo hàm từng phần $\partial E / c_i$ bằng 0 và giải hệ phương trình tuyến tính đối với c_i.

Khi xử lý những bài toán xấp xỉ, người ta thường gặp những ràng buộc khác như sau:

(1) Những ràng buộc nội suy:

$$g(x_j) = f(x_j) \text{ với một số điểm } x_j \text{ cố định;}$$

(2) Kết hợp điều kiện (1) với những điều kiện về độ trơn, ví dụ như điều kiện về đạo hàm của g và f đồng nhất tại điểm x_j;

(3) Các ràng buộc về tính trực giao

$$(g - f) \cdot \varphi_i = 0 \qquad \text{với mọi } i;$$

(4) Những ràng buộc về hình dạng trực quan, ví dụ như độ cong của đường cong và bề mặt.

Một đường cong tham số thường được định nghĩa như sau:

$$p:[a,b]\to R^m, \quad p(u)=(p_1(u),p_2(u),...,p_m(u))$$

với các hàm thành phần p_i của p là các hàm giá trị thực thông thường với một biến thực.

Định nghĩa. Nếu tất cả các hàm p_i đều có một tính chất nào đó thì ta nói rằng p cũng có tính chất đó. Ví dụ, nếu tất cả các hàm p_i đều là đa thức hoặc **spline** (khái niệm này sẽ được định nghĩa sau) thì ta nói rằng p là một đa thức hoặc một đường cong spline. Nếu tất cả các hàm p_i đều là đa thức tuyến tính, đa thức bậc hai hoặc bậc ba thì ta nói rằng p cũng là đường cong tuyến tính, đường cong bậc hai hoặc bậc ba.

8.2. Các đường cong tham số

8.2.1. Phép nội suy Lagrange

Dạng đơn giản nhất của phép nội suy là nội suy Lagrange.

Bài toán nội suy Lagrange: cho các điểm $(x_0,y_0),(x_1,y_1),...,$ và (x_n,y_n), tìm một đa thức $p(x)$, để $p(x_i)=y_i$ với $i=0,1,...,n$.

Chỉ có duy nhất một đa thức $p(x)$ bậc n như vậy, và được gọi là đa thức Lagrange. Trước hết, xem xét các đa thức:

$$L_{i,n}(x)=L_{i,n}(x;x_0,x_1,...,x_n)=\prod_{j=0,j\neq i}^{n}\frac{x-x_j}{x_i-x_j} \qquad (8.2)$$

Các đa thức $L_{i,n}$ được gọi là các *hàm cơ sở Lagrange* tương ứng với $x_0,x_1,...,x_n$. Đa thức Lagrange được xác định như sau:

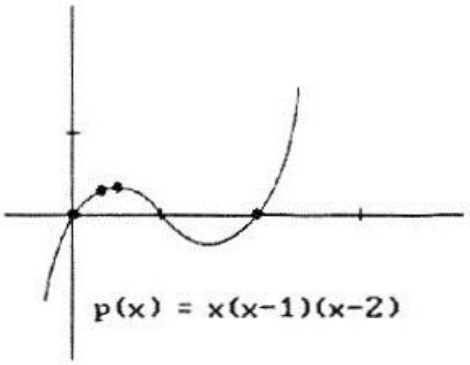
$$p(x)=\sum_{i=0}^{n}y_iL_{i,n}(x) \qquad (8.3)$$

Mặc dù luôn tồn tại đa thức để nội suy, có một vài hạn chế lớn khi sử dụng phương pháp này. Thứ nhất, bậc của đa thức sẽ lớn khi n lớn, và như vậy sẽ tốn nhiều chi phí tính toán. Thứ hai, đường cong tạo ra sẽ có những vết gợn không mong muốn khiến cho hình dáng của nó không giống như hình dáng mà dữ liệu mang lại. Ví

dụ, đa thức bậc ba duy nhất nội suy các điểm (0,0), (1/3,10/27), (1/2,3/8), và (2,0) là:

$$p(x) = x(x-1)(x-2)$$

nhưng hình dáng của nó lại ngoằn nghèo hơn hình dáng tạo ra bằng cách nối các điểm (Xem Hình 8.1). Không thể khử các vết gợn này vì một đa thức bậc n luôn có thể có đến n-1 các cực đại và cực tiểu.



Hình 8.1. Các vết gợn không mong muốn khi sử dụng đa giác nội suy.

8.2.2. Nội suy Hermite

Để tránh vấn đề dao động với nội suy Lagrange, người ta có thể nối các đa thức bậc 2 với nhau, tuy nhiên lại tạo nên các góc tại các điểm nối. Nếu thử dùng các đa thức bậc ba, ta sẽ có đủ số bậc tự do để ép các đa thức có cùng độ dốc tại các điểm nối. Với bốn ràng buộc cho trước là các số thực y_0, y_1, m_0 và m_1, tồn tại một đa thức bậc ba duy nhất thỏa mãn:

$$p(0) = y_0, \qquad p(1) = y_1, \qquad p'(0) = m, \text{ và } \quad p'(1) = m_1.$$

Thật vậy, với một đa thức bậc ba:

$$p(x) = a + bx + cx^2 + dx^3$$

có bốn bậc tự do, ta có thể thay các ràng buộc vào để tạo ra 4 phương trình 4 ẩn và sẽ thu được một đáp số duy nhất cho a, b, c và d. Chúng ta cũng có thể tiếp cận bằng ma trận. Vì

$$p'(x) = b + 2cx + 3dx^2,$$

ta có

$$\begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 \\ 3 & 2 & 1 & 0 \end{bmatrix} \begin{bmatrix} d \\ c \\ b \\ a \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ m_0 \\ m_1 \end{bmatrix} \tag{8.4}$$

Ma trận vuông 4x4 ở bên trái Công thức (8.2) có ma trận nghịch đảo:

$$M_h = \begin{bmatrix} 2 & -2 & 1 & 1 \\ -3 & 3 & -2 & -1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \tag{8.5}$$

Do đó:

$$\begin{bmatrix} d \\ c \\ b \\ a \end{bmatrix} = M_h \begin{bmatrix} y_0 \\ y_1 \\ m_0 \\ m_1 \end{bmatrix} \tag{8.6}$$

và

$$p(x) = \begin{pmatrix} x^3 & x^2 & x & 1 \end{pmatrix} M_h \begin{bmatrix} y_0 \\ y_1 \\ m_0 \\ m_1 \end{bmatrix} \mathfrak{M} \tag{8.7}$$

Nếu ta định nghĩa các đa thức F_1, F_2, F_3 và F_4 như sau:

$$(F_1(x)\ F_2(x)\ F_3(x)\ F_4(x)) = (x^3\ x^2\ x\ 1)M_h \tag{8.8}$$

thì

$$\begin{aligned} F_1(x) &= (x-1)^2(2x+1), \\ F_2(x) &= x^2(3-2x), \\ F_3(x) &= (x-1)^2 x, \text{ và} \\ F_4(x) &= x^2(x-1). \end{aligned} \tag{8.9}$$

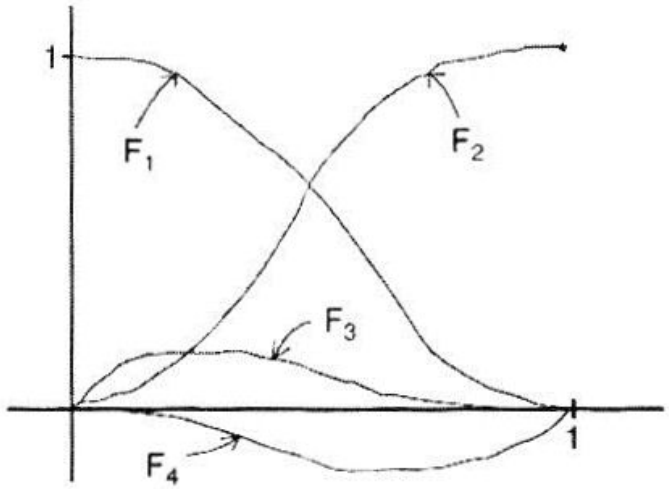
Quan trọng hơn,

$$p(x) = y_0F_1(x) + y_1F_2(x) + m_0F_3(x) + m_1F_4(x) \tag{8.10}$$

Chú ý là đa thức $F_i(x)$ thỏa mãn

$$F_1(0)=1, F_1(1)=0, F_1'(0)=0, F_1'(1)=0,$$
$$F_2(0)=0, F_2(1)=1, F_2'(0)=0, F_2'(1)=0,$$
$$F_3(0)=0, F_3(1)=0, F_3'(0)=1, F_3'(1)=0,$$
$$F_4(0)=0, F_4(1)=0, F_4'(0)=0, F_4'(1)=1 \qquad \textbf{(8.11)}$$

Hình 8.2 Biểu diễn đồ thị của các hàm đa thức này.



Hình 8.2. Các hàm cơ sở Hermite

Định nghĩa. Ma trận M_h được định nghĩa theo Công thức (8.5) được gọi là *ma trận Hermite*. Các đa thức $F_i(x)$ được định nghĩa theo Công thức (8.8) và (8.9) được gọi là các *hàm cơ sở Hermite*.

Lưu ý rằng:

$$F_1(x)+F_2(x)=1 \qquad \textbf{(8.12)}$$

với mọi x. Vì F_i cũng giống như các hàm cơ sở trong đa thức Lagrange, ta ký hiệu chúng như sau để tiện tham khảo:

$$H_{0,3}=F_1,\ H_{1,3}=F_3,\ H_{2,3}=F_4,\ H_{3,3}=F_2. \qquad \textbf{(8.13)}$$

Từ đây, ta có bài toán nội suy tổng quát:

Bài toán nội suy ghép đoạn (piecewise) Hermite: Cho các bộ ba (x_0,y_0,m_0), (x_1,y_1,m_1), ..., và (x_n,y_n,m_n), tìm các đa thức bậc ba $p_i(x), i=0,1,...,n-1$, để

$$p_i(x_i)=y_i,$$

$$p_i'(x_i) = m_i,$$
$$p_i(x_i + 1) = y_i, \text{ và}$$
$$p_i'(x_i + 1) = m_i + 1. \qquad \textbf{(8.14)}$$

Có thể chứng minh được bài toán nội suy ghép đoạn Hermite có một đáp số duy nhất.

8.2.3. Nội suy spline

Các bài toán nội suy và các hàm để giải quyết bài toán này như mô tả trong phần trước có thể được tổng quát hoá.

Định nghĩa. Một *đường spline bậc*[1] m *và cấp*[2] $m+1$ là một hàm $S:[a,b] \to R$ mà tồn tại các số thực x_i, $i = 0,...,n$ với $a = x_0 \le x_1 \le ... \le x_n = b$, để

(1) S là một đa thức có bậc $\le m$ trên đoạn $[x_i\ x_{i+1}]$, với $i = 0,...,n-1$ và

(2) S là một hàm C^{m-1}.

x_i được gọi là các *điểm nút* (*knot*) và $(x_0, x_1, ..., x_n)$ được gọi là các *vectơ điểm nút* có độ dài n+1 đối với đường spline. Các đoạn $[x_i, x_{i+1}]$ được gọi là các *nhịp* (*span*). Nếu một nút x_i thỏa mãn điều kiện $x_{i-1} < x_i = x_{i+1} = ... = x_{i+d-1} < x_{i+d}$ ($x_{-1} = -\infty$ và $x_{n+1} = +\infty$, thì x_i được coi là một *nút d bội*. S được gọi là đường spline tuyến tính, bậc hai hay bậc ba nếu nó có bậc là 1, 2 hay 3.

Các thuật ngữ chính có quan hệ mật thiết với khái niệm đường spline là "các hàm đa thức ghép đoạn", "nút" và "tính khả vi". Lưu ý rằng các đường spline không chỉ là các đa thức ghép đoạn mà chúng còn phải thỏa mãn điều kiện khả vi toàn cục. Hàm nội suy Hermite ghép đoạn được đề cập đến ở mục trước không đủ trơn để được coi là một đường spline bậc ba.

Định nghĩa vật lý của spline. Một spline là một đoạn kim loại hay gỗ có thể uốn cong để đi qua một số điểm cho trước.

---

[1] degree
[2] order

Các spline vật lý đã được sử dụng từ nhiều năm nay. Ví dụ trong việc kiến tạo vỏ ngoài của các con tàu, người ta xây dựng mô hình bằng cách tạo ra khung xương của than tàu. Nhiệm vụ này được các thợ tạo khung thực hiện bằng cách sử dụng các spline vật lý. Khi thử sử dụng toán học để mô tả các đường cong được tạo ra bởi các spline vật lý, người ta khám phá ra một điều rất thú vị. Tuân theo quy luật vật lý, các spline sẽ giữ hình dạng sao cho có thể giảm thiểu sức căng. Phương trình cho bài toán này khó có thể giải trực tiếp nhưng người ta có thể giải ra một cách xấp xỉ và kết quả là ta thu được một spline bậc ba. Với khả năng mô phỏng các đường spline vật lý cộng thêm yêu cầu tính toán thấp do bậc của chúng thấp làm cho spline bậc ba trở thành spline phổ biến nhất.

Bài toán nội suy spline: cho một số nguyên k và các số thực $\alpha = \beta = 3$ với $x_0 < x_1 < ... < x_n$, tìm spline $g(x)$ bậc k sao cho x_i là nút của g và $g(x_i) = y_i$.

Chúng ta có thể giải bài toán này với các đa thức bậc ba. Bài toán có thể được mô tả rõ hơn như sau: cho các điểm (x_0, y_0), (x_1, y_1), ..., (x_n, y_n), tìm các đa thức bậc ba $p_i(x)$, để với i chạy từ 0 đến $n-1$, ta có:

$$p_i(x_i) = y_i,$$
$$p_i(x_{i+1}) = y_{i+1},$$

và với i chạy từ 1 đến n-1 ta có:

$$p_i'(x_i) = p_{i-1}'(x_i)$$
$$p_i''(x_i) = p_{i-1}''(x_i)$$

Trong trường hợp này ta có 4n bậc tự do và chỉ có 4n-2 ràng buộc. Hai bậc tự do tự do còn lại có thể dùng để chọn m_0 và m_n, tương ứng với hệ số góc ở đầu và cuối spline. Chúng ta giới thiệu bốn cách chọn, tuy nhiên vẫn còn nhiều cách khác nữa.

Cách lựa chọn điều kiện kết thúc của các đường spline nội suy:

(1) (*Điều kiện kết thúc kẹp*) Ta có thể xác định các hệ số góc m_0 và m_n một cách rõ ràng.

(2) (*Điều kiện kết thúc Bessel*) Ta có thể đặt m_0 và m_n tương ứng là hệ số góc của đường parabôn nội suy ba điểm đầu và ba điểm cuối.

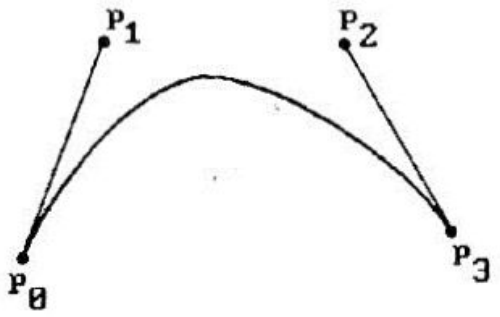
(3) (*Điều kiện kết thúc tự nhiên*) Chúng ta có thể muốn đạo hàm bậc hai của đường spline triệt tiêu ở các đầu mút, có nghĩa là gần thẳng ở điểm mút giống như trường hợp của một đường spline vật lý. Kiểu spline này được gọi là đường spline tự nhiên (Halics).

(4) (*Điều kiện kết thúc lặp*) Chúng ta cũng có thể muốn giá trị cũng như đạo hàm bậc 1 và bậc 2 của đường spline bằng nhau tại cả 2 đầu mút, tạo thành một đường spline đóng.

8.2.4. Các đường cong Bézier

Mục này và mục tiếp theo sẽ đề cập đến các đường cong được xác định bằng các điểm điều khiển nhưng nói chung không nội suy chúng.

Mặc dù việc sử dụng các hệ số hình học để xác định các đường cong là một cải tiến lớn so với việc xác định các hệ số đại số, các hệ số hình học này dưới dạng các vec-tơ tiếp tuyến vẫn mang nhiều tính kỹ thuật. Một phương pháp tốt để người dùng xác định các vec-tơ này một cách rõ ràng là chọn các điểm cho biết hình dạng cần có. Hình 8.3 cho thấy một đường cong bậc ba p(u) bắt đầu từ điểm p_0 và kết thúc ở điểm p_3.



Hình 8.3. Một đường cong Bézier.

Rất dễ xác định đường thẳng tiếp tuyến với đường cong tại hai điểm đầu mút bằng cách chọn bất kỳ điểm p1, p2 nào dọc theo các đường này. Như thế

$$p'(0) = \alpha p_0 p_1,$$
$$p'(1) = \beta p_2 p_3, \qquad \textbf{(8.15)}$$

với α và β nào đó. Bây giờ ta hãy xoay quanh cấu trúc này. Thay vì bắt đầu với đường cong *p(u)*, chúng ta sẽ bắt đầu với các điểm p_i và xét xem đường cong *p(u)* nào được xác định bởi các điểm này và bởi Công thức (8.15). Dĩ nhiên ta có thể gán cho α và β bất kỳ giá trị số thực dương nào đó, nhưng trước hết ta sẽ đặt cố định $\alpha = \beta = 3$.

Theo định nghĩa, ma trận Hermite B_h của đường cong bậc ba $p(u)$ là $M_{hb}B_b$, với:

$$M_{hb} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ -3 & 3 & 0 & 0 \\ 0 & 0 & -3 & 0 \end{bmatrix}$$

và

$$B_b = \begin{bmatrix} p_0 \\ p_1 \\ p_2 \\ p_3 \end{bmatrix}$$

Ma trận B_b chính là dữ liệu hình học của đường cong Bézier. Nó thỏa mãn:

$$p(u) = UM_h B_b = UM_h M_{hb} B_b = UM_b B_b = F_b B_b \qquad \textbf{(8.16)}$$

trong đó $F_b = UM_b$ và

$$M_b = M_h M_{hb} = \begin{pmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix}$$

Định nghĩa. Các thành phần của ma trận B_b, cụ thể là các điểm p_i, được gọi là hệ số Bézier của *đường* cong p(u). Ma trận Bb được gọi là ma trận Bézier.

Nhân các ma trận trong Công thức (8.16) sẽ cho ta công thức sau đây:

$$p(u) = (1-u)^3 p_0 + 3u(1-u)^2 p_1 + 3u^2(1-u)p_2 + u^3 p_3. \quad \textbf{(8.17)}$$

Đường cong bậc 3 này được gọi là đường cong Bézier dựa vào các điểm p_i. Lưu ý cách các tổng các hệ số bằng 1, có nghĩa là đường cong nằm trong bao lồi của các điểm p_i. Điều này giải thích cho việc chọn $\alpha = \beta = 3$ trong Công thức (8.15).

Như vậy, chúng ta đã có một cách thức mới để thiết lập đường cong. Chọn 4 điểm tạo ra một đường cong sao cho đường cong đó bắt đầu từ điểm đầu tiên và kết thúc ở điểm cuối cùng, có vec-tơ tiếp tuyến ở 2 đầu mút tương ứng song song với các đường thẳng nối giữa hai điểm đầu và hai điểm cuối. Ta điều chỉnh đường cong bằng cách dịch chuyển một hoặc vài "điểm điều khiển" này.

Tiếp đó, chúng ta sẽ tổng quát hóa việc tạo đường cong từ 4 điểm thành tạo đường cong từ một số lượng điểm bất kỳ. Chúng ta mong muốn có thể xác định một đường con bằng cách phác họa ra hình dáng cần có của nó bằng một số điểm. Chúng ta sẽ mô tả 2 cách tiếp cận để xác định các đường cong Bézier tổng quát:

(1) Khởi đầu với xấp xỉ đa thức Bernstein của các hàm liên tục và tiếp tục "**dốc sức**" (*brute-force*) để có được một số thuộc tính, hoặc

(2) Áp dụng cách tiếp cận "đa affine" mang nhiều tính hình học hơn.

Chúng ta bắt đầu với cách thứ nhất.

Định nghĩa. Cho $f: [0, 1] \to R^m$ là một hàm liên tục. Định nghĩa

$$F_n(f)(u) = \sum_{i=0}^{n} f(\frac{i}{n}) B_{i,n}(u),$$

trong đó:

$$B_{i,n}(u) = \binom{n}{i} u^i (1-u)^{n-i}.$$

Hàm đa thức $F_n(f)(u)$ được gọi là *xấp xỉ đa thức Bernstein bậc n* đối với f.

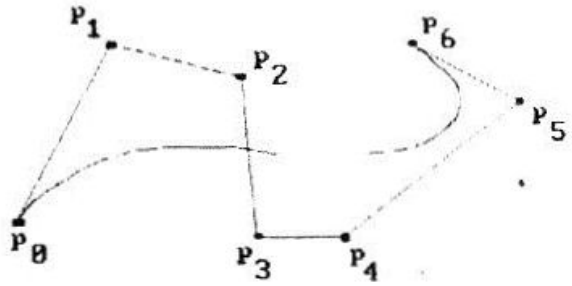
Bernstein đã sử dụng những đa thức này để đưa ra một chứng minh một cách xây dựng cho định lý xấp xỉ Weierstrass, chỉ ra rằng tất cả các hàm liên tục có thể gần xấp xỉ bằng một đa thức. Người ta cũng chỉ ra rằng các đa thức Berstein hội tụ đều về f, nhưng chúng hội tụ rất chậm và vì thế không thường được sử dụng để hội tụ trong toán học vì có nhiều cách tốt hơn để xấp xỉ các hàm bằng đa thức. Tuy nhiên, chúng có dùng để biểu diễn các đường cong thuận tiện trong thiết kế đường cong một cách tương tác.

Định nghĩa. Cho trước một dãy các điểm p_i thuộc R^m, với $i = 0,1,2,\ldots,n$, định nghĩa một đường cong Bézier $p(u)$ với $u \in [0,\ 1]$ bằng phương trình

$$p(u) = \sum_{i=0}^{n} B_{i,n}(u) p_i.$$

Các điểm p_i được gọi là các điểm Bézier hay các điểm điều khiển của p(u) và đường cong đa giác được xác định bởi các điểm này được gọi là tạo nên đa giác Bézier hay đa giác đặc trưng hay đa giác điều khiển của p(u). Các hàm $B_{i,n}(u)$ được gọi là các hàm cơ sở Bézier.

Đường cong Bézier có một số thuộc tính rất tiện lợi. Trước hết, các đường cong Bézier nằm trong bao lồi của đa giác đặc trưng. Thứ hai, đường cong Bézier bắt đầu ở điểm điều khiển đầu tiên và kết thúc ở điểm điều khiển cuối cùng, có nghĩa là, $p(0) = p_0$ và $p(1) = p_n$. Thứ ba, các đường Bézier có *tính đối xứng*. Điều này có nghĩa là nếu ta liệt kê danh sách các điểm điều khiển của một đường cong theo thứ tự ngược lại thì ta cũng nhận được một đường cong y hệt đi theo hướng ngược lại.



Hình 8.4. Một đường cong Bézier.

Ưu điểm của đường cong Bézier so với đường cong Hermite là xác định 4 điểm thì có vẻ trực quan hơn là 2 điểm và 2 tiếp tuyến. Hơn thế nữa, thực tế là đường cong nằm trong bao lồi của các điểm điều khiển làm cho việc cắt xén dễ hơn. Trước hết, người ta tiến hành cắt xén phần bao lồi với cửa sổ cắt. Nếu chúng không cắt nhau thì đường cong sẽ ở bên ngoài cửa sổ.

Có 2 vấn đề với đường cong Bézier tổng quát là:

(1) Bậc của đường cong tăng cùng với số lượng các điểm điều khiển.

(2) Không có khả năng điều khiển cục bộ. Thay đổi của bất kỳ điểm điều khiển nào cũng buộc phải tính toán lại toàn bộ đường cong, mặc dù sự thay đổi càng nhỏ đối với những điểm ở càng xa điểm điều khiển bị thay đổi.

Cách thông thường để tránh nhược điểm đầu tiên là sử dụng đường cong phân đoạn Bézier, nhưng sau đó thì ta lại phải giải quyết vấn đề làm mịn toàn bộ đường cong và liệu từng đoạn riêng rẽ đó có khớp nhau thành đường cong một cách tốt đẹp không. Có vài thủ thuật mà ta có thể áp dụng để lưu giữ cảm giác mịn của đường cong. Ví dụ, bằng cách mỗi lần lấy cả 4 điểm điều khiển, ta có thể thu được một đường cong Bézier bậc ba phân đoạn. Nếu cạnh cuối của một đa giác điều khiển ứng với một bộ 4 điểm song song với cạnh đầu của đa giác điều khiển của bộ 4 điểm tiếp theo, đường cong sẽ mịn mà không có một gấp khúc rõ rệt nào. Để đạt được điều này ta có thể bổ sung thêm các điểm điều khiển vào tập hợp ban đầu, chúng có thể là các điểm giữa tại các đoạn thích hợp. Hình 8.5(a) cho thấy hai đường cong Bézier 4 điểm gặp nhau ở đầu mút với góc cạnh rõ rệt. Bằng cách bổ sung thêm các điểm p và q,

ta thu được một đường cong mới như trong Hình 8.5(b), là kết hợp của ba đường cong Bézier 4 điểm mà không có góc cạnh nào.

Hình 8.5. Các đường cong Bézier

8.2.5. Đường cong B-Spline

Một vấn đề chung của các đường cong đã được giới thiệu là bất kỳ thay đổi nào với một điểm điều khiển cũng buộc ta phải tính toán lại toàn bộ đường cong. Đây là điều hoàn toàn không mong muốn. Các đường cong B-Spline sẽ giúp giải quyết vấn đề này. Thay đổi các điểm điều khiển sẽ chỉ có ảnh hưởng cục bộ đến đường cong mà thôi.

Đường B-Spline được định nghĩa theo rất nhiều cách. Ta có thể định nghĩa chúng bằng

(1) một cách tiếp cận "**dốc sức**" (*brute-force*) bằng cách giải phương trình mô tả những ràng buộc nhất định.

(2) hàm lũy thừa cắt gọn (hàm lũy thừa một phía),

(3) đệ quy,

(4) ma trận (thường dùng trong trường hợp đường B-Spline bậc hai và bậc ba) hoặc

(5) cách tiếp cận ánh xạ đa affine.

Chúng ta sẽ xét tất cả những phương pháp này, ngoại trừ phương pháp thứ 2 vì khởi đầu bằng cách xét một trường hợp đặc biệt sẽ giúp làm rõ và góp phần thúc đẩy những bàn thảo tổng quát phía sau.

Các đa giác là những ví dụ đơn giản nhất của đường cong B-Spline, mặc dù thường thì người ta không nghĩ thế. Vì chúng đơn giản nên rất hữu dụng trong việc làm rõ một số khía cạnh cơ bản của đường B-Spline. Xét một đa giác X với các đỉnh $p_i = (x_i, y_i)$, trong đó $i = 0,1,...,n$. Ta định nghĩa các hàm S_i và S như sau:

$$S_i(t) = (1-t)y_i + t\,y_{i+1}$$

và

$$S(t) = \begin{cases} y_0 & \text{nếu } t < x_0 \\ S_i((t-x_i)/(x_{i+1}-x_i)) & \text{nếu } x_i \le t \le x_{i+1} \\ y_0 & \text{nếu } x_n < t \end{cases}$$

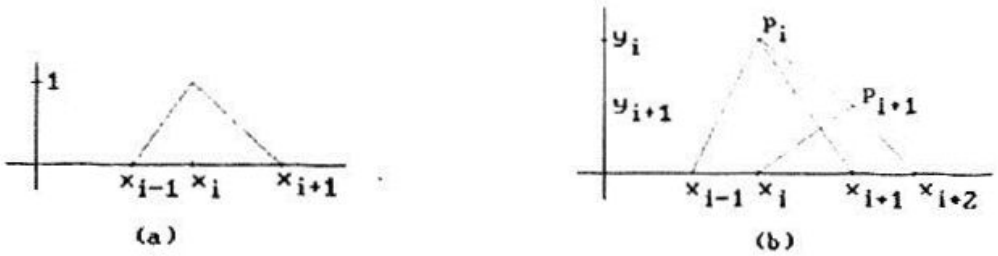
Như vậy, hàm S là một hàm spline với các nút $t_i = x_i$ và đa giác là một đồ thị của hàm này trên đoạn $[x_0, x_n]$. Đồ thị của S_i trên đoạn $[x_i, x_{i+1}]$ chính là đoạn $X_i = [p_i, p_{i+1}]$ của X và S_i tương ứng với biểu diễn tuyến tính của X_i.

Lưu ý rằng nếu ta dịch bất kỳ điểm p_i nào thì chỉ có 2 hàm liền nhau là S_{i-1} và S_i là bị ảnh hưởng. Chúng ta có thể viết S như là tổng của các hàm cơ sở mà có thể địa phương hóa những thay đổi tương tự như những gì ta đã thấy trong trường hợp của nội suy Lagrange và nội suy Hermite. Cụ thể, với mỗi i xét một đơn vị hàm "hình mũ" $b_i(t)$ được xác định bằng

$$b_i(t) = \begin{cases} 0 & \text{nếu } t < x_{i-1} \\ (t-x_{i-1})/(x_i - x_{i-1}) & \text{nếu } x_{i-1} \le t \le x_i \\ (x_{i+1}-t)/(x_{i+1}-x_i) & \text{nếu } x_i \le t \le x_{i+1} \\ 0 & \text{nếu } x_{i+1} < t \end{cases}$$

Trong Hình 8.6(a). b_i là các B-Spline tuyến tính vì chúng là các đường spline khác không trên chỉ 2 đoạn. Một tính chất của các hàm "hình mũ" này, mà dường như lúc này không mang ý nghĩa quan trọng, là tổng của chúng bằng 1, đó là,

$$\sum_{i=0}^{n} b_i(t) = 1 \qquad \text{với } x_1 \le t \le x_{n-1}$$



Hình 8.6 Các đường B-spline tuyến tính.

Mỗi đoạn của đa giác, ngoại trừ đoạn đầu và đoạn cuối, chính là đồ thị của hàm tổng của 2 hàm "hình mũ". Chính xác hơn,

$$S_i(t) = y_i b_i(t) + y_{i+1} b_{i+1}(t) \qquad \text{với } 0 < i < n$$

Xem Hình 8.6(b). Nói cách khác, nếu ta tránh các điểm mút thì

$$S_i(t) = \sum_{i=0}^{n} y_i b_i(t) \qquad \text{với } x_i < t < x_{n-1}.$$

Sử dụng B-Spline tuyến tính cho ta một đường cong liên tục. Nếu ta muốn một đường cong mịn hơn, ta cần đến những hàm cơ sở bậc cao hơn. Các B-Spline bậc hai sẽ cho ta những đường cong khả vi. B-Spline bậc ba sẽ cho ta đường cong khả vi bậc hai. Sau đây là những tính chất mà một B-Spline bậc m phải có:

(1) Phải là một đường spline bậc m.

(2) Phải triệt tiêu ngoài $m+1$ khoảng của B-spline.

Định lý sau sẽ cho chúng ta thấy phương pháp "**dốc sức**" (*brute-force*) có thể dùng để cho thấy các đường spline bậc ba thỏa mãn điều kiện triệt tiêu nói trên.

Định lý. Cho trước các nút khác nhau x_{i-2}, x_{i-1}, x_i, x_{i+1}, x_{i+2}, tồn tại một đường spline bậc ba $b_i(t)$ duy nhất thỏa mãn:

(1) $b_i(t) = 0$, với $t < x_{i-2}$ hoặc $x_{i+2} < t$ và

(2) $b_i(x_{i-1}) + b_i(x_i) + b_i(x_{i+1}) = 1$.

Định lý này có thể dễ dàng được chứng minh. Về cơ bản, chúng ta có 4 đa thức bậc ba, tương ứng với 4 đoạn, cho ta 16 bậc tự do. Về ràng buộc, chúng ta muốn các đa thức có cùng giá trị, đạo hàm bậc nhất và đạo hàm bậc hai ở 5 nút, tạo ra 15 ràng buộc (ở hai điểm mút x_{i-2} và x_{i+2}, các giá trị hàm và đạo hàm của chúng bằng 0). Thêm ràng buộc trong điều kiện (2) tạo ra một phương trình cho một kết quả duy nhất. Kết quả này gồm 4 đa thức:

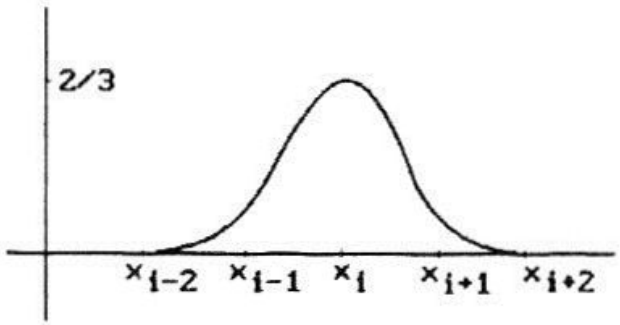
$$(1/6)u^3, \qquad u = \frac{t - x_{i-2}}{x_{i-1} - x_{i-2}},$$

$$(1/6)(-3u^3 + 3u^2 + 3u + 1), \qquad u = \frac{t - x_{i-1}}{x_i - x_{i-1}},$$

$$(1/6)(3u^3 - 6u^2 + 4), \qquad u = \frac{t - x_i}{x_{i+1} - x_i}, \text{ và}$$

$$(1/6)(-u^3 + 3u^2 - 3u + 1), \qquad u = \frac{t - x_{i+1}}{x_{i+2} - x_{i+1}}.$$

trong đó t tương ứng thuộc các đoạn sau $[x_{i-2}, x_{i-1}]$, $[x_{i-1}, x_i]$, $[x_{i-1}, x_{i+1}]$, $[x_{i+1}, x_{i+2}]$, (xem Hình 8.7). Không có điều kiện (2), ta sẽ có nhiều đáp số.



Hình 8.7. Một hàm B-spline cơ sở bậc ba.

Bây giờ chúng ta sẽ trình bày hàm B-spline theo định nghĩa đệ quy Cox-de Boor.

Định nghĩa. Cho trước $n \geq 0$, $k \geq 1$, và một dãy không giảm các số thực $U = (u_0, u_1, \ldots, u_{n+k})$, định nghĩa các hàm

$$N_{i,k} : R \to R, \quad 0 \leq i \leq n,$$

một cách đệ quy như sau:

$$N_{i,1}(u) = \begin{cases} 1, & \text{đối với } u_i \leq u \leq u_{i+1} \\ 0, & \text{trong trường hợp ngược lại.} \end{cases} \qquad \textbf{(8.18a)}$$

Nếu $k > 1$ thì:

$$N_{i,k}(u) = \frac{u - u_i}{u_{i+k-1} - u_i} N_{i,k-1}(u) + \frac{u_{i+k} - u}{u_{i+k} - u_{i+1}} N_{i+1,k-1}(u), \qquad \textbf{(8.18b)}$$

trong đó nếu bất kỳ đại lượng nào có dạng 0/0 thì ta sẽ thay nó bằng 0. Hàm $N_{i,k}(u)$ được gọi là **B-Spline thứ i** hoặc **hàm B-Spline cơ sở bậc** k **và cấp** $k-1$ **theo vectơ nút U**.

Lưu ý rằng một B-Spline bậc k không nhất thiết phải là một đường spline bậc k. Nếu có nút nào có bội số lớn hơn 1 thì đường B-Spline sẽ không đủ khả vi tại điểm này. Mặt khác, bất cứ đường spline nào cũng là tổ hợp tuyến tính của các hàm B-Spline cơ sở. Chính vì các B-Spline là cơ sở cho không gian spline nên người ta mới gọi chúng là B-spline với chữ B là *basis* – cơ sở.

Định nghĩa. Một vec-tơ nút của một đường spline hoặc một đường B-Spline bậc k được gọi là **bị kẹp** (*clamped*) nếu các nút đầu tiên và cuối cùng có bội số là k. Ngược lại, nó được gọi là **không bị kẹp**. Một véc-tơ nút (không bị kẹp) U có độ dài L thì được coi là **đồng nhất** (*uniform*) hoặc có **tuần hoàn** (*periodic*) nếu các nút u_i được đặt cách đều nhau, có nghĩa là, tồn tại một hằng số $d > 0$ để $u_{i+1} = u_i + d$ với $0 \le i \le L-2$. Nếu U là bị kẹp thì U được gọi là **đồng nhất** nếu tất cả các nút u_i ngoại trừ k nút đầu tiên và k nút cuối cùng được đặt cách đều nhau, có nghĩa là, $u_{i+1} = u_i + d$ với $k \le i \le L-K$. Một véc-tơ nút không có tính chất đồng nhất thì được gọi là **không đồng nhất**.

Định nghĩa. Các spline hoặc B-spline là **bị kẹp, không bị kẹp, đồng nhất, tuần hoàn** hoặc **không đồng nhất** nếu véc-tơ nút của nó là bị kẹp, không bị kẹp, đồng nhất, tuần hoàn hoặc không đồng nhất.

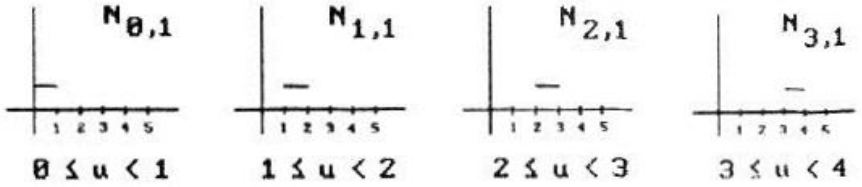
Định nghĩa. Cho $(u_0, u_1, \ldots, u_{n+k})$ là các vectơ nút của một đường B-spline bậc k. Nó được gọi là vec-tơ nút **đồng nhất chuẩn** (*standard uniform*) nếu $u_i = i$. Nó được gọi là vec-tơ **đồng nhất bị kẹp chuẩn** (*standard clamped uniform*) nếu

$$u_0 = u_1 = \ldots = u_{k-1} = 0$$
$$u_i = i - k + 1, \quad \text{với } k \le i \le n,$$
$$u_{n+1} = u_{n+2} = \ldots = u_{n+k} = n - k + 2$$

Sau đây chúng ta sẽ xem xét một số ví dụ về các hàm cơ sở B-spline với giả thiết rằng $0 \le u \le n-k+2$. Ví dụ. $n = 3, k = 1$, các nút u_i trong trường hợp này là

$$u_0 = 0; u_1 = 1; u_2 = 2; u_3 = 3; u_4 = 4$$

Hình 8.8 cho thấy đồ thị của $N_{i,1}(u)$.

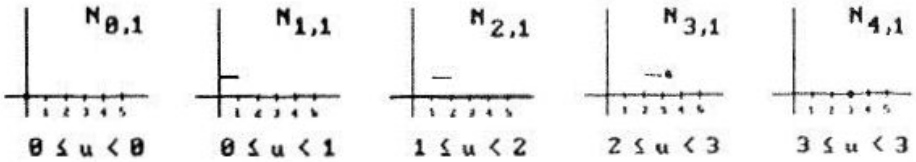


Hình 8.8. Ví dụ hàm $N_{i,1}(u)$ với n=3, k=1.

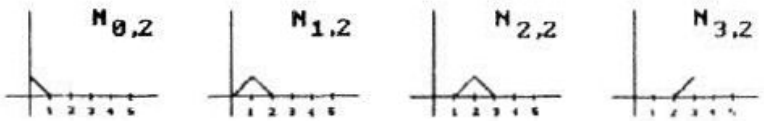
Ví dụ, với $n = 3, k = 1$, u_i sẽ là

$$u_0 = u_1 = 0; u_2 = 1; u_3 = 2; u_4 = u_5 = 4$$

Đồ thị của $N_{i,j}$ được thể hiện trong Hình 8.9.



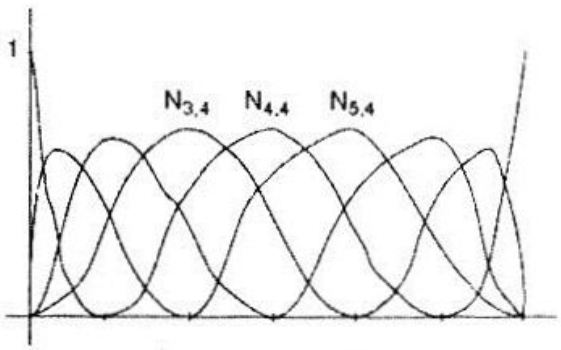
(a)



(b)

Hình 8.9. Một ví dụ các hàm Ni,j(u) với n=3, k=2.

Hình 8.10 cho thấy đồ thị của các B-spline bậc ba đồng nhất bị kẹp $N_{i,4}(u)$ với $n = 8$.



Hình 8.10. Các B-spline bậc ba đồng nhất bị kẹp $N_{i,4}(u)$ với n = 8.

Định lý. Các hàm $N_{i,k}(u)$ được định nghĩa bởi Công thức (8.18) thỏa mãn các tính chất sau:

(1) (*Tính chất hỗ trợ rút gọn*) Hàm $N_{i,k}(u)$ triệt tiêu trên $(-\infty, u_i) \cup [u_{i+k}, \infty)$. Cụ thể, chỉ các hàm $N_{i-k+1}(u), N_{i-k+2}(u), \ldots, N_{i,k}(u)$ là khác 0 trên $[u_i, u_{i+1})$.

(2) (*Tính khả vi*) Hàm $N_{i,k}(u)$ là C^{∞} bên trong các đoạn và C^{k-1-m} tại nút có bội m. Cụ thể, $N_{i,k}(u)$ là một spline bậc k nếu tất cả các nút có bội là 1.

(3) $N_{i,k}(u) \geq 0$ với mọi u.

(4) Đồng nhất thức

$$\sum_{i=0}^{n} N_{i,k}(u) = 1$$

đúng với mọi $u \in [k-1, k+1]$ và không đúng cho bất cứ u nào khác. Nếu vec-tơ nút là bị kẹp, thì đồng nhất thức sẽ đúng với mọi $u \in [0, n+k]$.

Tính chất hỗ trợ rút gọn của đường B-spline rất quan trọng vì nó có nghĩa là chúng ta có thể tạo sửa đổi các đường cong một cách cục bộ mà không phải tính toán lại toàn bộ đường cong.

Định nghĩa. Cho trước một dãy các điểm p_i, với $i = 0, 1, \ldots, n$, đường cong

$$p(u) = \sum_{i=0}^{n} N_{i,k}(u) p_i. \qquad (8.19)$$

được gọi là **đường cong B-Spline bậc** k (hoặc *cấp* $m = k-1$) với các **điểm điều khiển** hay **các điểm Boor** $\mathbf{p_i}$ và **vec-tơ nút** $(u_0, u_1, .., u_{n+k})$. Nếu các véc-tơ nút có tính chất **bị kẹp**, **không bị kẹp**, **đồng nhất**, **tuần hoàn** hoặc **không tuần hoàn** thì đường cong cũng có tính chất tương tự. **Miền** của đường cong này được xác định là đoạn đóng $[u_{k-1}, u_{n+1}]$. (Lưu ý rằng nếu đường cong là bị kẹp, thì miền này gồm toàn bộ đoạn $[u_0, u_{n+k}]$.) Mỗi phần $p(u_i, u_{i+1})$, $k-1 \leq i \leq n$, của toàn bộ đường cong xác định bởi

$p(u)$ được gọi là một **đoạn** (*segment*) của đường cong. Đường cong đa giác được xác định nhờ các điểm điều khiển được gọi là **đa giác Boor** hay **đa giác điều khiển** của đường cong.

Cuối cùng, chúng ta sẽ liệt kê một số nhận xét về đường cong B-spline.

(1) Nếu đường B-spline là bị kẹp thì nó nội suy các điểm điều khiển đầu và cuối.

(2) Việc chèn thêm các điểm điều khiển vào cùng một chỗ buộc đường cong phải đi qua điểm điều khiển này vì như vậy sẽ làm tăng bội của nút.

(3) Tăng bậc của một đường cong B-spline với một lượng cố định các điểm điều khiển sẽ làm giảm số đoạn của đường cong, và như vậy, làm giảm sự ảnh hưởng của bất kỳ điểm điều khiển nào với đường cong.

(4) Với một đường B-Spline không tuần hoàn, chúng ta phải có $n \geq k - 1$. Nói cách khác, ta phải có ít nhất số điểm điều chỉnh bằng số bậc của đường spline. Ta cần có $n \geq k$ nếu ta muốn có một đường spline với miền lớn hơn 1 điểm.

(5) Các đường Bézier được coi như trường hợp đặc biệt của đường cong B-Spline. Ví dụ, 4 đường B-Spline từ đường spline tuần hoàn trên một dãy các nút 0, 0, 0, 0, 1, 1, , 1 cho ta các đa thức Bernstein bậc ba.

8.3. Các bề mặt tham số

Phần này mô tả các bề mặt tham số và bề mặt ẩn chính mà ta gặp trong đồ họa máy tính và trong mô hình hóa hình học. Đây là các bề mặt mà ta thường đề cập tới như các bề mặt được **gọt giũa** (*sculptured*) hoặc các bề mặt **dạng tự do** (*free-form*) để phân biệt chúng với các bề mặt **tuyến tính từng đoạn** (*piecewise linear*). Các bề mặt ở đây chính là sự mở rộng của những gì ta đã xây dựng đối với đường cong trong phần trước.

Tiếp diện của các bề mặt, đặc biệt là pháp tuyến của chúng, đóng vai trò rất quan trọng như tiếp tuyến đối với đường cong. Trong trường hợp tham số, thì mặt phẳng tham số sẽ được xác định

từ đạo hàm từng phần liên quan đến biểu diễn tham số. Trong trường hợp bề mặt ẩn, chúng được xác định nhờ phương trình.

Có nhiều kiểu bề mặt trơn và có nhiều cách khác nhau để biểu diễn những bề mặt này. Một số cách biểu diễn liên quan đến nội suy dữ liệu, một số khác chỉ xấp xỉ dữ liệu. Người ta thường phân chia các bề mặt theo ba nguyên tắc biểu diễn bề mặt: **vạch mẫu** (*lofting*), **chồng đống** (*superposition*) và thể hiện qua tích Tensor hay tích Cartesian. Các **bề mặt được quy tắc hóa** (*ruled surface*), các bề mặt Coons, và các bề mặt Bézier hoặc B-Spline là những ví dụ minh họa cho 3 nguyên tắc này.

Vạch mẫu đôi khi được coi như một trường hợp đặc biệt của cách xây dựng bề mặt tổng quát hơn được gọi là thể hiện thông qua **sinh đường chuẩn** (*directrix-generator*). Đây là cách biểu diễn một bề mặt bằng cách quét một đường cong sinh theo những hướng dẫn nhất định. Ba thành phần cho một biểu diễn như vậy là:

(1) một tập các đường cong theo chiều dọc được gọi là **đường chuẩn**- (*directrices*) (hay còn gọi là **kinh tuyến** trong toán học),

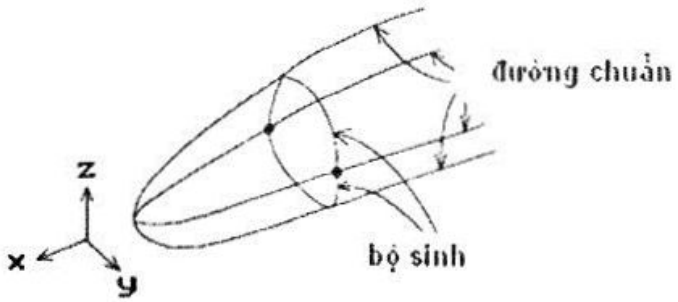
(2) một **quy tắc tương ứng** kết nối từng điểm của đường chuẩn với một điểm duy nhất nằm trên tất cả các đường chuẩn khác, và

(3) một **quy tắc sinh** xác định một đường cong đi qua tất cả các điểm trên đường chuẩn được kết nối bởi quy tắc tương ứng.

Hình 8.11 cho ta thấy ví dụ về việc một bề mặt được xác định như thế nào nhờ kiểu cấu trúc sinh đường chuẩn. Nó là một ví dụ truyền thống về cách vạch mẫu các đường conic của một thân máy bay. Các đường chuẩn là đường conic và bộ sinh gồm 2 đường conic, một ở phía trên đường thẳng chiều rộng lớn nhất và một ở phía dưới. Quy tắc tương ứng kết nối các điểm có cùng giá trị x.

Rất nhiều bề mặt có thể được xác định bởi cách xây dựng bằng sinh đường chuẩn. [Sabi90] chỉ ra một số ví dụ. Cụ thể, các bề mặt được quy tắc hóa và các **bề mặt tròn xoay** (*surfaces of revolution*) rõ ràng là thuộc loại các bề mặt này. Đường chuẩn của bề mặt được quy tắc hóa là các đường cong bao ngoài và bộ sinh là các đoạn

thẳng nối các điểm tương ứng trên những đường cong này. Một bề mặt tròn xoay có duy nhất một đường chuẩn, chính là đường cong được xoay, và các bộ sinh là các đường tròn. Ý tưởng về một biểu diễn sinh từ đường chuẩn có thể được tổng quát hóa để cho phép các đường chuẩn hoạt động như các điểm điều khiển chứ không chỉ là được nội suy.



Hình 8.11. Một thân máy bay được tạo ra từ bề mặt sinh đường chuẩn (directrix generator).

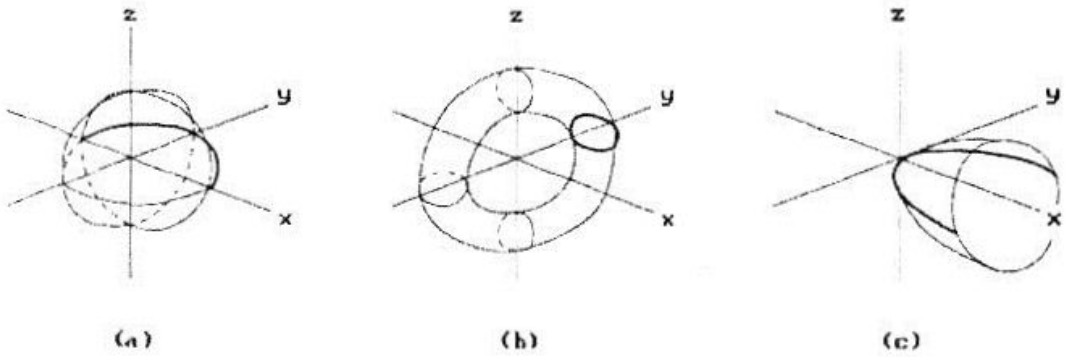
Mục đích chính của việc có nhiều kiểu bề mặt khác nhau là để cho người dùng và các lập trình viên dễ dàng xây dựng bề mặt mà họ cần. Đưa ra những mô tả về tập các điểm 3 chiều rõ ràng là khó hơn là tập các điểm trên mặt phẳng. Một trong những lý do hiển nhiên là ta đang cố gắng mô tả 3 chiều trên một thiết bị 2 chiều - màn hình máy tính. Để làm cho các chương trình dựng mô hình hình học trở nên thân hiện với người dùng, ta muốn tránh việc bắt người dùng phải nhập các phương trình toán học phức tạp. Một hướng tiếp cận thông thường để thực hiện điều này là để người dùng xác định bề mặt từ tập thành phần ít chiều hơn bằng cách sử dụng các thao tác tự nhiên. Ví dụ để xác định một bề mặt tròn xoay thì người dùng chỉ cần xác định được đường cong để xoay tròn và trục quay.

8.3.1. Các bề mặt tròn xoay

Các **bề mặt tròn xoay** (*surfaces of revolution*) là một dạng bề mặt thường gặp. Bề mặt cầu và bề mặt trụ có thể coi là thuộc dạng này. Thông thường, người ta tạo nên một "đối tượng tròn xoay" bằng việc xoay một tập quanh một trục bất kỳ. Một ví dụ đơn giản là khi xoay một điểm, ta được một đường tròn nằm trên mặt phẳng

vuông góc với trục quay và có bán kính bằng với khoảng cách từ điểm đó đến trục quay. Từ đó, ta có thể coi một đối tượng tròn xoay là một loạt các đường tròn có tâm nằm trên trục tương ứng với một điểm của đối tượng được xoay tròn. Điều này cũng gợi ý cho ta một cách tham số hóa điểm p của một đối tượng nhận được khi ta xoay một đường cong quanh một trục, đó là sử dụng 2 tham số. Một tham số là tham số của điểm trên đường cong và tham số kia là góc quay. Đối với hầu hết các đối tượng tròn xoay ta cần có $k+1$ tham số, trong đó k là số tham số cần để tham số hóa đối tượng xoay.

Để đơn giản hóa, chúng ta sẽ xét việc tham số hóa trong trường hợp một đường cong được xoay tròn quanh trục x. Sau đó, ta có thể có được các bề mặt tròn xoay quanh một trục bất kỳ thông qua phép quay và tịnh tiến. Ngoài ra, ta cũng giả thiết rằng đường cong nằm trên mặt phẳng x-y. Cũng nên lưu ý rằng dù dễ dàng xác định một đối tượng tròn xoay nhưng không hề dễ dàng khi đảm bảo kết quả sẽ cho ra một bề mặt. Kể cả trong một vài trường hợp đặc biệt mà ta sẽ phân tích, cũng không thể bảo đảm rằng kết quả sẽ không có điều gì bất thường.



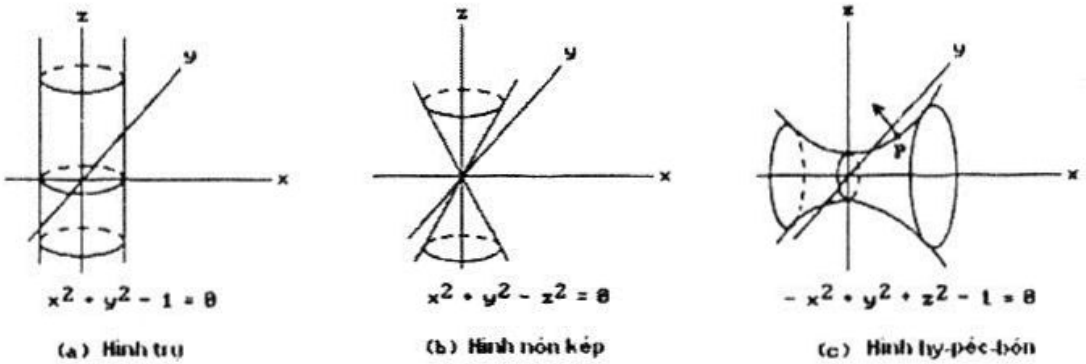
Hình 8.12. Các bề mặt tròn xoay: bề mặt cầu, bề mặt xuyến và bề mặt parabol.

8.3.2. Các bề mặt bậc hai

Cũng giống như đường cong bậc hai, các bề mặt bậc hai là những hình dạng quan trọng đối với thiết kế đồ họa. Một bề mặt bậc hai là một tập các điểm (x, y, z) trong $\mathbf{R}^3$ thỏa mãn phương trình bậc hai tổng quát có dạng

$$ax^2 + by^2 + cz^2 + dxy + exz + fyz + gx + hy + iz + j = 0 \quad \textbf{(8.20)}$$

Bỏ qua các trường hợp suy biến, ta sẽ thu được các bề mặt elip, bề mặt trụ, bề mặt nón, bề mặt parabol và bề mặt hyperbol cơ bản. Hình 8.12(a) và Hình 8.12(c) cho thấy một bề mặt cầu (trường hợp đặc biệt của elip) và một bề mặt parabol. Hình 8.13 cho thấy một số ví dụ về bề mặt bậc hai khác. Rõ ràng, nhiều bộ phận cơ khí có hình dạng tương tự thế này, và chúng là những ứng cử viên tốt cho các mẫu cơ bản trong mô hình hóa.



Hình 8.13. Ba bề mặt bậc hai.

8.3.3. Các bề mặt theo quy tắc

Các bề mặt theo quy tắc có lẽ là các bề mặt đơn giản thứ hai sau mặt phẳng. Một trường hợp đặc biệt của chúng là các **bề mặt trồi** (*extrusions*). Chúng là những bề mặt nhận được bằng cách quét một véc-tơ dọc theo một đường cong.

Định nghĩa. Cho một đường cong $f: [a,b] \to \mathbf{R}^3$ và vec-tơ $v \in \mathbf{R}^3$, bề mặt tham số

$$p: [a,b]\mathrm{X}[0,1] \to \mathbf{R}^3$$

được định nghĩa bởi

$$p(u, t) = f(u) + tv \qquad \textbf{(8.21)}$$

được gọi là một bề mặt trồi. Véc-tơ v được gọi là véc-tơ quét của bề mặt trồi.

Hình 8.14. Các bề mặt trôi và bề mặt vạch mẫu.

Định nghĩa. Cho trước 2 đường cong f và g: $[a, b] \to \mathbf{R}^3$, bề mặt tham số

$$p: [a,b]\mathrm{X}[0,1] \to \mathbf{R}^3$$

được xác định bởi

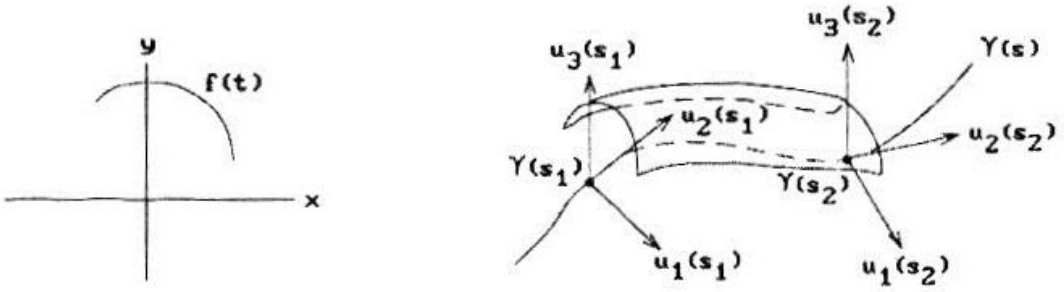
$$p(u, v) = (1 - v) f(u) + vg(u) \qquad (8.22)$$

được gọi là một bề mặt **vạch mẫu**.

Các bề mặt **vạch mẫu** là các bề mặt theo quy tắc mà nội suy ra 2 đường cong. Lưu ý rằng bề mặt trôi là một trường hợp đặc biệt của bề mặt **vạch mẫu**. Bề mặt trụ và bề mặt nón chính là những bề mặt theo quy tắc.

8.3.4. Các bề mặt quét

Các bề mặt quét là có thể được coi như bao ngoài của khối hình mà ta có được khi quét một tập (đường cong hoặc khối hình) dọc theo một đường cong, ví dụ như trong Hình 8.15. Theo nghĩa này thì bề mặt tròn xoay và bề mặt theo quy tắc là các bề mặt quét.



Hình 8.15. Quét một đường cong dọc theo một đường cong khác.

8.3.5. *Các bề mặt song tuyến tính*

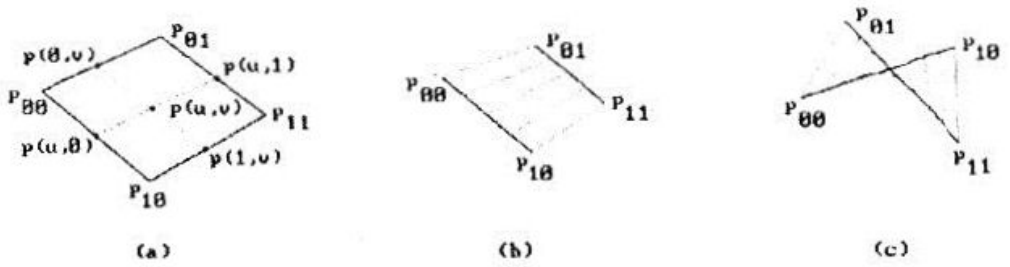
Cho 4 điểm p_{00}, p_{01}, p_{10} và p_{11}, cách dễ nhất để xác định một bề mặt p(u,v) nội suy các điểm này là sử dụng một phép nội suy tuyến tính kép. Định nghĩa

$$p(u,v) = (1-v)\left[(1-u)p_{00} + u.p_{10}\right] + v\left[(1-u)p_{01} + u.p_{11}\right], \tag{8.23a}$$

$$= (1-u)\left[(1-v)p_{00} + v.p_{01}\right] + u\left[(1-v)p_{10} + v.p_{11}\right], \tag{8.23b}$$

$$= (1-u)(1-v)p_{00} + (1-u)v.p_{01} + u(1-v)p_{10} + u.vp_{11}.$$

trong đó $0 \le u,\ v \le 1$. Công thức (8.23a) cho biết để tìm $p(u,v)$, trước hết ta phải tìm $p(u,0)$ và $p(u,1)$ nhờ phép nội suy tuyến tính theo hướng u, sau đó là thực hiện nội suy tuyến tính của các điểm theo hướng v. Công thức (8.23b) cho thấy ta sẽ nhận được kết quả tương tự nếu ta nội suy theo hướng v trước và sau đó là nội suy theo hướng u. Xem hình 12.10(a).



Hình 8.16. Các bề mặt song tuyến tính.

Định nghĩa. Một bề mặt tham số được định nghĩa nhờ Công thức (8.23) được gọi là bề mặt song tuyến tính được xác định bằng 4 điểm hoặc bề mặt nội suy bốn điểm.

Bề mặt song tuyến tính là trường hợp đặc biệt của bề mặt **vạch mẫu** và cũng là trường hợp đặc biệt của bề mặt Coons và bề mặt tích tensor Bézier được mô tả ở phần sau. Ta có thể viết lại hàm p(u,v) dưới dạng ma trận như sau:

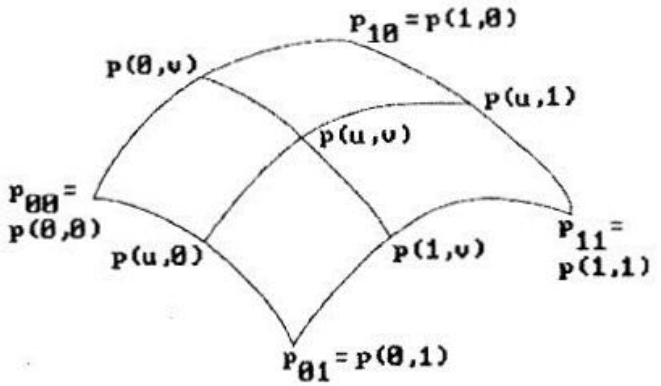
$$p(u,v) = (1-u, u)\begin{pmatrix} p_{00} & p_{01} \\ p_{10} & p_{11} \end{pmatrix}\begin{pmatrix} 1-v \\ v \end{pmatrix}$$

Nếu các điểm p_i là đồng phẳng và độc lập tuyến tính (như trong Hình 8.16(b)) thì bề mặt kết quả sẽ là tứ giác, nếu không (như trong Hình 8.16(c)) thì nó là một bề mặt bậc hai.

8.3.6. Các bề mặt Coons

Các bề mặt mà chúng đã đề cập thường được mô tả rất đơn giản chỉ bằng một công thức. Có nhiều bề mặt khác mà ta cần trong thiết kế đồ họa, như thân xe hơi hoặc thân máy bay, thì không thể mô tả như vậy. Trong những trường hợp này thì giải pháp chung là chia nhỏ bề mặt ra làm nhiều mảnh, được xác định lưới các đường cong. Mỗi mảnh được tham số hóa bằng cách "đổ đầy" hoặc nội suy các biểu diễn tham số của các đoạn biên của nó. Bản thân các đoạn cong biên thường được tham số hóa bằng cách nội suy dữ liệu thu nhận được của hình cần mô hình hóa.

Trong mục này ta sẽ mô tả một vài phương pháp tổng quát để xác định các bề mặt tương ứng với dữ liệu biên cho trước. Trong một số tài liệu, các biểu diễn tham số như vậy đôi khi được gọi là các phép nội suy siêu hạn vì ta nội suy một tập vô hạn các điểm. Tất cả các bài toán nội suy mà ta đã bàn luận trước đây (ngoại trừ trường hợp bề mặt **vạch mẫu**) chỉ nội suy một tập hữu hạn các điểm. Ngoài ra, dữ liệu trên đường biên liên quan không chỉ đến các cong biên mà còn liên quan đến vấn đề các mảnh được ghép khít vào nhau một cách mịn màng.

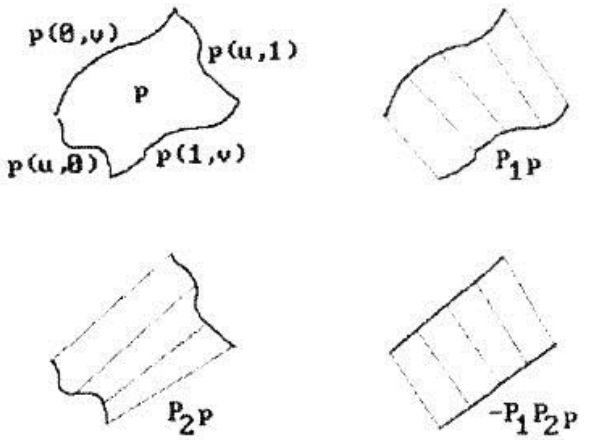


Hình 8.17. Một mảnh của bề mặt cong Coons.

Nêu rõ mục tiêu một cách cụ thể hơn: cần tìm một biểu diễn tham số $p(u,v)$ với $u,v \in [0,1]$, bằng cách nội suy 4 đường cong ngoại biên cho trước $p(0,v)$, $p(1,v)$, $p(u,0)$, $p(u,1)$ xem Hình 8.17). Coons đã đưa ra một cách giải quyết rất nổi tiếng cho bài toán này [Coon67]. Cách tiếp cận của ông được xây dựng dựa trên biểu diễn tham số của bề mặt **vạch mẫu** đối với các cặp đường cong ngoại biên $p(u,0)$, $p(u,1)$ và $p(0,v)$, $p(1,v)$. Xác định các toán tử P_1 và P_2 trên hàm 2 biến $p(u,v)$ bởi

$$(P_1 p)(u,v) = (1-u)\,p(0,v) + up(1,v) \qquad \textbf{(824a)}$$

$$(P_2 p)(u,v) = (1-v)\,p(u,0) + vp(u,1) \qquad \textbf{(824b)}$$



Hình 8.18. Một mảnh Coons song tuyến tính trộn.

So sánh những công thức của $P_1 p$ và $P_2 p$ với Công thức (8.22). Các hàm này nội suy các đường cong biên theo hướng v và u. Nếu các đường cong biên là đường thẳng được tham số hóa bằng phương pháp tuyến tính thông thường thì ta sẽ thu được bề mặt song tuyến tính được xác định bằng các Công thức (8.2.) và $P_1 p(u,v)$ và $P_2 p(u,v)$ trở thành một hàm. Nếu các đường cong biên không phải là đường thẳng thì $P_1 p(u,v)$ và $P_2 p(u,v)$ khác nhau và chúng cũng không nội suy ra các đường cong biên khác theo hướng u và v (xem Hình 8.18). Ví dụ, sự khác biệt giữa $P_1 p$ và các giá trị thực tế của p theo các biên hướng u là

$$p(u,0) - P_1 p(u,0) \quad \text{và} \quad p(u,1) - P_1 p(u,1)$$

Nội suy đại lượng lỗi này theo hướng v thu được $P_2(p - P_1p)(u,v)$. Nếu ta định nghĩa

$$p(u,v) = P_1p(u,v) + P_2(p - P_1p)(u,v)$$

$$= P_1p(u,v) + P_2p(u,v) - P_2P_1p(u,v).$$

(8.25a)

thì việc xây dựng hàm $p(u,v)$ sẽ nội suy ra toàn bộ biên. Nếu ta áp dụng những công thức vừa rồi cho P_2p và tính sai khác với giá trị thực tế của p dọc theo các biên hướng v, ta cũng sẽ thu được cùng một công thức (8.25a). Như ta thấy trong Hình 8.18, $P_1P_2(u,v)$ chính là một bề mặt theo quy tắc kép. Thay thế các toán tử P_i trong công thức bằng $p(u,v)$ trong Công thức (8.25a) bằng định nghĩa của chúng dẫn đến công thức Coons nguyên gốc.

$$p(u,v) = (1-v)p(u,0) + vp(u,1) + (1-u)p(0,v) + up(1,v)$$

$$-(1-u)(1-v)p(0,0) - (1-u)vp(0,1) - u(1-v)p(1,0) - uvp(1,1).$$

(8.25b)

Định nghĩa. Bề mặt tham số được xác định bởi hàm $p(u,v)$ trong Công thức (**8.25**) được gọi là mảnh Coons hoặc bề mặt Coons (song tuyến tính trộn) đối với các đường cong $p(0,v)$, $p(1,v)$, $p(u,0)$ *và* $p(u,1)$.

Bề mặt Coons có thể biểu diễn dưới dạng ma trận như sau:

$$p(u,v) = (1-u\ \ u)\begin{pmatrix} p(0,v) \\ p(1,v) \end{pmatrix} + (1-v\ \ v)\begin{pmatrix} p(u,0) \\ p(u,1) \end{pmatrix} - (1-u\ \ u)\begin{pmatrix} p(0,0) & p(0,1) \\ p(1,0) & p(1,1) \end{pmatrix}\begin{pmatrix} 1-v \\ v \end{pmatrix}$$

$$= (1-u\ \ u\ \ 1)\begin{pmatrix} -p(0,0) & -p(0,1) & p(0,v) \\ -p(1,0) & -p(1,1) & p(1,v) \\ p(u,0) & p(u,1) & 0 \end{pmatrix}\begin{pmatrix} 1-v \\ v \\ 1 \end{pmatrix}$$

Ta có thể dễ dàng tổng quát hóa bề mặt Coons cơ bản bằng cách thay thế các hàm trộn tuyến tính đơn giản bằng các hàm khác. Xét $b_0(t)$, $b_1(t)$, $c_0(t)$ và $c_1(t)$ là các hàm giá trị thực bất kỳ

trên đoạn $[0,1]$ và định nghĩa các toán tử mới P_1 và P_2 trên các hàm $p(u,v)$ bởi

$$(P_1p)(u,v) = b_0(u)\,p(0,v) + b_1(u)\,p(1,v) \tag{8.26a}$$

$$(P_2p)(u,v) = c_0(v)\,p(u,0) + c_1(v)\,p(u,1) \tag{8.26b}$$

Lưu ý là định nghĩa của các hàm $(P_ip)(u,v)$ chỉ dùng các giá trị biên của $p(u,v)$. Các Công thức mới này sẽ được rút gọn thành Công thức (8.24) nếu ta đặt $b_0(t) = c_0(t) = L_{0,1}(t)$ và $b_1(t) = c_1(t) = L_{1,1}(t)$, trong đó $L_{0,1}(t) = 1 - t$ và $L_{1,1}(t) = t$ là các hàm cơ sở Lagrange tuyến tính. Thực ra, để các hàm được xác định nhờ Công thức (8.26) nội suy biên theo hướng v và u, hàm trộn $b_i(t)$ và $c_i(t)$ chỉ cần thỏa mãn điều kiện

$$b_0(0)=1, \quad b_0(1)=0, \quad b_1(0)=0, \quad b_1(1)=1,$$

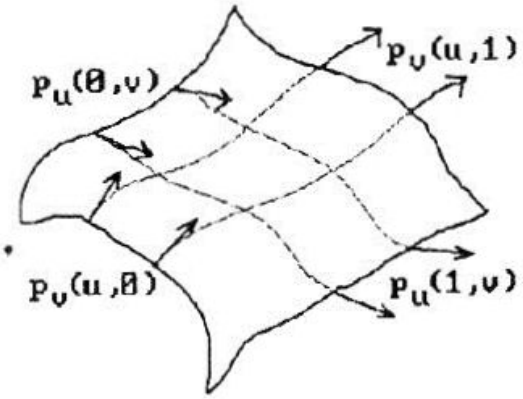
$$c_0(0)=1, \quad c_0(1)=0, \quad c_1(0)=0, \quad c_1(1)=1.$$

Ta có thể thu được một bề mặt Coon mịn hơn nhờ sử dụng các hàm cơ sở Hermite $H_{i,3}$ được giới thiệu trong Công thức **8.13** của Mục 8.2.2 và định nghĩa

$$p(u,v) = (H_{0,3}(u)\;\; H_{3,3}(u)\;\; 1)\begin{pmatrix} -p(0,0) & -p(0,1) & p(0,v) \\ -p(1,0) & -p(1,1) & p(1,v) \\ p(u,0) & p(u,1) & 0 \end{pmatrix}\begin{pmatrix} H_{0,3}(v) \\ H_{3,3}(v) \\ 1 \end{pmatrix} \tag{8.27}$$

Định nghĩa. Bề mặt tham số được xác định bởi hàm p(u,v) trong Công thức (**8.27**) được gọi là *mảnh* hay *bề mặt Coons song lập phương (bicubic)* cho các đường cong $p(0,v)$, $p(1,v)$, $p(u,1)$, $p(u,0)$ và $p(u,1)$.

Với một lưới các đường cong cho trước, bề mặt nội suy tổng thể, mà ta có được bằng biểu diễn tham số Coons song tuyến tính trộn cho mỗi mảnh tách biệt, có thể chỉ là C^0 dù cho bản thân các đường cong là C^1. Vấn đề này là do việc sử dụng các hàm trộn tuyến tính. Một trong số những tính chất rất hay của bề mặt Coons song lập phương là nó cho ta những bề mặt mịn tổng thể. Có nghĩa là nếu ta dùng các bề mặt Coons song lập phương thì ta sẽ có được một bề mặt C^1 tổng thể.



Hình 8.19. Một bề mặt Coons song lập phương mịn.

Mặc dù các mảnh Coons song lập phương cho ta một bề mặt C^1 nếu các đường cong biên là C^1, chúng ta không có nhiều kiểm soát với các đạo hàm dọc theo các biên. Để linh hoạt hơn, giả thiết rằng ta được cho trước các đạo hàm từng phần $p_u(0,v)$, $p_u(1,v)$, $p_v(u,0)$, và $p_v(u,1)$ (xem Hình 8.19). Định nghĩa các toán tử mới Q_1 và Q_2 là

$$Q_1 p(u,v) = H_{0,3}(u)\,p(0,v) + H_{1,3}(u)\,p_u(0,v) + H_{2,3}(u)\,p_u(1,v) + H_{3,3}(u)\,p(1,v) \quad \textbf{(8.28a)}$$

$$Q_2 p(u,v) = H_{0,3}(v)\,p(u,0) + H_{1,3}(v)\,p_v(u,0) + H_{2,3}(v)\,p_v(u,1) + H_{3,3}(v)\,p(u,1) \quad \textbf{(8.28b)}$$

và định nghĩa

$$Qp = (Q_1 + Q_2 - Q_1Q_2)p \quad \textbf{(8.29)}$$

Đặt

$$B=\begin{pmatrix} p(0,0) & p(0,1) & p_v(0,0) & p_v(0,1) \\ p(1,0) & p(1,1) & p_v(1,0) & p_v(1,1) \\ p_u(0,0) & p_u(0,1) & p_{uv}(0,0) & p_{uv}(0,1) \\ p_u(1,0) & p_u(1,1) & p_{uv}(1,0) & p_{uv}(1,1) \end{pmatrix}$$

Công thức (**8.29**) có thể được viết lại dưới dạng

$$Qp(u,v) = (b_0(u)\ b_1(u)\ d_0(u)\ d_1(u)\ 1)\begin{pmatrix} & & & & p(0,v) \\ & -B & & & p(1,v) \\ & & & & p_u(0,v) \\ & & & & p_u(1,v) \\ p(u,0) & p(u,1) & p_v(u,0) & p_v(u,1) & 0 \end{pmatrix}\begin{pmatrix} c_0(v) \\ c_1(v) \\ e_0(v) \\ e_1(v) \\ 1 \end{pmatrix} \quad \textbf{(8.30)}$$

Định nghĩa. Bề mặt tham số $Qp(u,v)$ được xác định nhờ Công thức (**8.29**) và dạng ma trận của nó (**8.30**) được gọi là *mảnh* hay *bề mặt Coons song lập phương tổng quát.*

Bề mặt Coons song lập phương tổng quát, ngoài tính chất C^1 tổng thể, còn có thể kiểm soát với các đạo hàm dọc theo các biên.

8.3.7. Các bề mặt tích tensor

Các bề mặt tích tensor là một trong những bề mặt hay gặp nhất trong thiết kế đồ họa. Một vài dạng đơn giản có thể được tính toán bằng ma trận. Trước hết, chúng ta sẽ xem xét một cách tổng quan. Các vấn đề chi tiết liên quan đến những trường hợp đặc biệt sẽ được đề cập ở các mục con.

Xét một đường cong

$$p(u)=\sum_{i=0}^{m} f_i(u)p_i.$$

trong đó $f_i(u)$ là các hàm cơ sở, và ta coi p_i là họ các hàm một tham số với giá trị véc-tơ

$$p_i(v)=\sum_{j=0}^{n} g_j(v)p_{ij}.$$

với các hàm cơ sở $g_j(v)$ và các điểm p_{ij}.

Định nghĩa. Bề mặt tham số $p(u,v)$ xác định bởi

$$p(u,v)=\sum_{i=0}^{m}\sum_{j=0}^{n}f_i(u)g_j(v)p_{ij}. \qquad \textbf{(8.31)}$$

được gọi là bề mặt tích Tensor hay bề mặt tích Đề-các với các hàm cc sở $f_i(u)g_j(v)$.

Dưới dạng ma trận thì Công thức (**8.31**) biến thành

$$p(u,v)=(f_0(u)\ f_1(u)\dots f_m(u))\begin{pmatrix} p_{00} & \cdot & \cdot & p_{0n} \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ p_{m0} & \cdot & \cdot & p_{mn} \end{pmatrix}\begin{pmatrix} g_0(v) \\ g_1(v) \\ \cdot \\ g_n(v) \end{pmatrix} \qquad \textbf{(8.32)}$$

Tiếp theo, chúng ta sẽ thảo luận một số bề mặt tích Tensor thông dụng: mảnh song lập phương, các bề mặt Bézier, các bề mặt B-Spline và các bề mặt B-Spline **hữu tỷ** (*rational*).

8.3.7.1. Mảnh song lập phương

Định nghĩa. Một bề mặt tham số $p(u,v)$ được xác định bởi

$$p(u,v)=\sum_{i=0}^{3}\sum_{j=0}^{3}u^i v^j p_{ij}, \qquad \textbf{(8.33)}$$

trong đó $0\le u, v\le 1$, được gọi là *bề mặt tích tensor song lập phương tổng quát* hay *mảnh song lập phương*. Tương tự với trường hợp của đường cong, các điểm p_{ij} được gọi là các hệ số đại số của mảnh song lập phương.

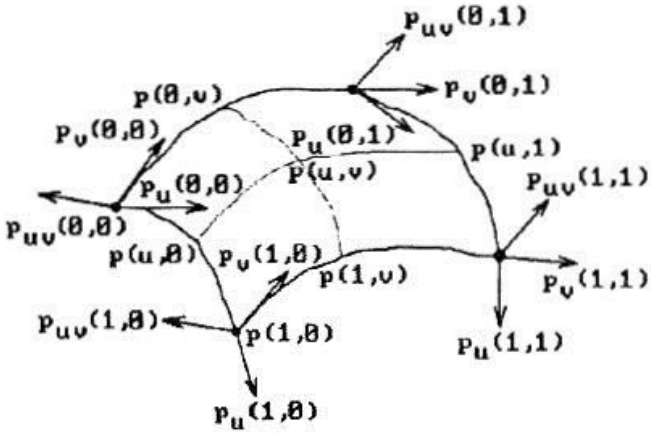
Dễ dàng thấy rằng mảnh song lập phương có thể được xem như là tích Tensor của 2 đường cong lập phương. Cũng giống như trong trường hợp các đường cong, mô tả đại số (**8.33**) thường không thuận tiện lắm với người dùng. Người ta muốn có một cách thức mang tính hình học hơn để xác định bề mặt. Một cách để thu được mô tả hình học là theo quan sát sau. Có 16 (véc-tơ) bậc tự do. Một số ràng buộc hình học rõ ràng là 4 điểm góc và 8 vec-tơ tiếp tuyến theo hướng u và v ở các điểm này. Còn lại 4 bậc tự do và ta có thể

sử dụng các đạo hàm từng phần hỗn hợp $p_{uv}(u,v)$ tại các điểm góc. Chúng được gọi là các *véc-tơ xoắn* (twist vector).

Định nghĩa. Ma trận B được xác định bởi

$$B = \begin{pmatrix} p(0,0) & p(0,1) & p_v(0,0) & p_v(0,1) \\ p(1,0) & p(1,1) & p_v(1,0) & p_v(1,1) \\ p_u(0,0) & p_u(0,1) & p_{uv}(0,0) & p_{uv}(0,1) \\ p_u(1,0) & p_u(1,1) & p_{uv}(1,0) & p_{uv}(1,1) \end{pmatrix}$$

được gọi là *ma trận hình học* của mảnh song lập phương. Các thành phần của nó được gọi là các hệ số hình học của mảnh.



Hình 8.20. Sử dụng ma trận hình học để tính p(u,v).

Các hệ số đại số hoàn toàn có thể được xác định qua ma trận hình học (xem Hình 8.20):

(1) $p(u,0)$ được xác định từ $p(0,0)$, $p(1,0)$, $p_u(0,0)$ và $p_u(1,0)$.

(2) Tương tự, $p(u,1)$ được xác định từ $p(0,1)$, $p(1,1)$, $p_u(0,1)$, và $P_u(1,1)$.

(3) Tiếp đó, $p_v(u,0)$ được xác định từ $p_v(0,0)$, $p_v(1,0)$, $p_{uv}(0,0)$, và $p_{uv}(1,0)$.

(4) Tương tự, $p_v(u,1)$ được xác định từ $p_v(0,1)$, $p_v(1,1)$, $p_{uv}(0,1)$, và $p_{uv}(1,1)$.

(5) Cuối cùng, $p(u,v)$ được xác định từ $p(u,0)$, $p(u,1)$, $p_v(u,1)$, và $P_v(u,1)$.

8.3.7.2. Các bề mặt Bézier

Đối với nhiều người, việc xác định các tiếp tuyến và đặc biệt là các vec-tơ xoắn đối với mảnh song lập phương không thực sự trực giác. Trong trường hợp này, phương pháp Bézier có thể được sử dụng, tuy nhiên lần này chúng ta cần xác định một lưới 16 điểm p_{ij} (xem Hình 8.21). Đặt

$$B_b = \begin{pmatrix} p_{00} & p_{01} & p_{02} & p_{03} \\ p_{10} & p_{11} & p_{12} & p_{13} \\ p_{20} & p_{21} & p_{22} & p_{23} \\ p_{30} & p_{31} & p_{32} & p_{33} \end{pmatrix}$$

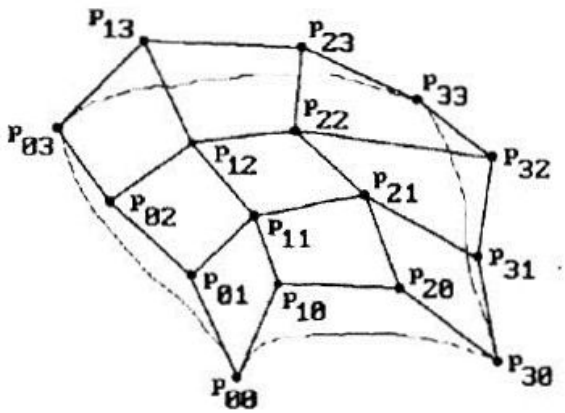
và định nghĩa một biểu diễn tham số $p(u,v)$, $0 \le u,v \le 1$, bằng

$$p(u,v) = UM_b B_b M_b{}^T V^T \qquad \textbf{(8.34)}$$

trong đó

$$M_b = \begin{pmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix}$$

Định nghĩa. Bề mặt tham số $p(u,v)$ được gọi là một bề mặt Bézier lập phương. Các thành phần của ma trận B_b, cụ thể là các điểm p_{ij}, được gọi là các hệ số Bézier của bề mặt Bézier này.



Hình 8.21. Một bề mặt Bézier lập phương.

Sau đây là một vài tính chất của bề mặt Bézier, tương tự như các tính chất của đường cong Bézier.

(1) Các đường cong biên của bề mặt Bézier là đường cong Bézier.

(2) Chỉ nội suy các đỉnh góc, tuy nhiên hình dạng của bề mặt rất sát với các điểm điều khiển p_{ij}.

(3) Các vec-tơ $p_{00}p_{10}$ và $p_{00}p_{01}$ tạo nên tiếp diện tại p_{00}, tương tự với các điểm góc khác.

(4) Mảnh Bézier nằm trong bao lồi của các điểm điều khiển.

8.3.7.3. Các bề mặt B-Spline

8.3.7.3.1. Các bề mặt B-Spline cơ bản

Xét $N_{i,k}(u)$ và $N_{j,h}(v)$ là các hàm được xác định bằng Công thức **8.18** với các véc-tơ nút không giảm cho trước $(u_0, u_1, \ldots, u_{m+k})$ và $(v_0, v_1, \ldots, v_{n+h})$. Xét p_{ij} là tập các điểm đã cho. Xác định một hàm $p(u,v)$ bởi

$$p(u,v) = \sum_{i=0}^{m}\sum_{j=0}^{n} N_{i,k}(u) N_{j,h}(v) p_{ij}. \qquad \textbf{(8.35)}$$

Định nghĩa. Bề mặt tham số p(u,v) được xác định bởi Công thức **(8.35)** được gọi là một bề mặt B-Spline bậc[3] (k,h) và cấp[4] $(k-1, h-1)$ với các điểm điều khiển p_{ij} và nút theo u là u_i, nút theo v là a v_j. Miền xác định của bề mặt được xác định là khoảng vuông $[u_{k-1}, u_{m+1}]\mathrm{X}[v_{h-1}, v_{n+1}]$. Nếu $k = h = 3$, thì bề mặt được gọi là bề mặt B-Spline song lập phương.

Bề mặt tích tensor B-Spline được xác định bởi công thức (8.16) thỏa mãn một số tính chất quan trọng kéo theo từ các tính chất của đường cong tương ứng. Ví dụ,

(1) (Điều khiển địa phương) Nếu một điểm p_{ij} bị di chuyển thì chỉ biến đổi hàm $p(u,v)$ trong khoảng $[u_i, u_{i+k}) \times [v_j, v_{j+h})$.

---

[3] order

[4] degree

(2) Tại một nút u với bội số r, các đạo hàm từng phần $\partial^i p / \partial u_i$ tồn tại và liên tục với $0 \le i \le k-1-r$. Tại một nút v của bội số s, đạo hàm từng phần $\partial^j p / \partial v_j$ tồn tại và liên tục với $0 \le j \le h-1-s$.

(3) (Tính chất bao lồi địa phương) Các bề mặt B-spline thỏa mãn tính chất bao lồi, có nghĩa là chúng trong bao lồi của các điểm điều khiển. Trong thực tế, giống như trong trường hợp đường cong B-spline, sẽ có một tính chất mạnh hơn tồn tại: nếu $(u,v) \in [u_i, u_{i+1}) \times [v_j, v_{j+1})$ thì $p(u,v)$ nằm trong bề mặt lồi của các điểm p_{st}, $i-k+1 \le s \le i$, $j-h+1 \le t \le j$.

(4) Nếu các đoạn cong B-spline $N_{i,k}(u)$ và $N_{j,h}(v)$ có các vectơ nút bị kẹp thì $p(u,v)$ nội suy 4 điểm góc, đó là:

$p(u_{k-1}, v_{h-1}) = p_{00}$, $p(u_{m+1}, v_{h-1}) = p_{m0}$, $p(u_{k-1}, v_{n+1}) = p_{0n}$, và $p(u_{m+1}, v_{n+1}) = p_{mn}$.

(5) Nếu $m = k-1$, $n = h-1$, $(u_i) = (0, \ldots, 0, 1, \ldots, 1)$ và $(v_j) = (0, \ldots, 0, 1, \ldots, 1)$, thì $p(u,v)$ xác định một bề mặt Bézier.

8.3.7.3.2. Các bề mặt B-spline hữu tỷ

Các bề mặt B-spline hữu tỷ là các bề mặt tương ứng với đường cong B-spline hữu tỷ và mục đích sử dụng chúng cũng tương tự, cụ thể là mặc dù các bề mặt B-spline đã rất tổng quát rồi, nhưng chúng không bao hàm cả các bề mặt bậc 2 mà chỉ có thể xấp xỉ chúng. Với việc sử dụng các đường cong và bề mặt B-spline hữu tỷ, một hệ thống mô hình hóa chỉ cần phải hỗ trợ một dạng biểu diễn đồng nhất cho các đối tượng hình học của nó.

Xét các bề mặt tích Tensor được xác định bởi Công thức (8.12). Chúng là các bề mặt trong không gian affine. Bằng cách sử dụng tọa độ đồng nhất, các bề mặt tương tự trong không gian chiếu của chúng có dạng:

$$P(u,v) = \sum_{i=0}^{m}\sum_{j=0}^{n} a_i(u)b_j(v)P_{ij}, \tag{8.36}$$

trong đó P_{ij} là các điểm đã được mô tả với tọa độ đồng nhất. Diễn giải các điểm P_{ij} theo dạng $(w_{ij}x_{ij},\ w_{ij}y_{ij},\ w_{ij}z_{ij},\ w_{ij})$, bề mặt trong không gian chiếu được xác định bởi $P(u,v)$ sẽ chiếu lên bề mặt

$$p(u,v) = \frac{\sum_{i=0}^{m}\sum_{j=0}^{n} a_i(u)b_j(v)w_{ij}p_{ij}}{\sum_{i=0}^{m}\sum_{j=0}^{n} a_i(u)b_j(v)w_{ij}} \tag{8.37}$$

trong đó $p_{ij} = (x_{ij}, y_{ij}, z_{ij})$.

Định nghĩa. Bề mặt tham số $p(u,v)$ được xác định bởi Công thức (8.37) được gọi là một *bề mặt tích tensor hữu tỷ*. Nó còn được gọi là *bề mặt Bézier hữu tỷ* nếu miền xác định của nó là $[0,1]\times[0,1]$ và $a_i(u) = B_{i,m}(u)$ và $b_j(v) = B_{j,n}(v)B_{st}(u)$ là các hàm được xác định bởi đường cong *Bézier*:

$$B_{i,n}(u) = \binom{n}{i} u^i (1-u)^{n-i}$$

Bề mặt $p(u,v)$ được gọi là *bề mặt B-Spline hữu tỷ* bậc (k,h) và cấp $(k-1,\ h-1)$ nếu $a_i(u)$ và $b_j(v)$ là các B-Spline bậc k và h. Các nút $a_i(u)$ được gọi là *các nút theo u* (*u-knots*) của bề mặt và các nút $b_j(v)$ là *các nút theo v* của bề mặt. Đối với cả 2 trường hợp Bézier và B-Spline, các điểm p_{ij} được gọi là các *điểm điều khiển* của bề mặt và w_{ij} được gọi là các *trọng số*.

Định nghĩa. Các hàm

$$R_{ij}(u,v) = \frac{a_i b_j(v)w_{ij}}{\sum_{s=0}^{m}\sum_{t=0}^{n} a_s b_t(v)w_{st}} \tag{8.38}$$

được gọi là các *hàm cơ sở hữu tỷ* của bề mặt xác định bởi Công thức (8.37).

Sử dụng các hàm cơ sở hữu tỷ, Công thức (**8.37**) có thể được viết lại như sau:

$$p(u,v)=\sum_{i=0}^{m}\sum_{j=0}^{n}R_{ij}(u,v)p_{ij}. \tag{8.39}$$

Kiểu thông dụng nhất của bề mặt B-Spline hữu tỷ được mô tả như sau:

Định nghĩa. Nếu các B-Spline $N_{i,k}(u)$ và $N_{j,h}(v)$ (xác định bởi các Công thức (**8.37**)) được xác định với các vectơ nút như sau:

$$U=(\underbrace{a,\ldots,a}_{k},u_k,u_{k+1},\ldots,u_m,\underbrace{b,\ldots,b}_{k})$$

và $$U=(\underbrace{c,\ldots,c}_{h},v_h,v_{h+1},\ldots,v_n,\underbrace{d,\ldots,d}_{h})$$

thì bề mặt B-Spline hữu tỷ

$$p(u,v)=\frac{\sum_{i=0}^{m}\sum_{j=0}^{n}N_{i,k}(u)N_{j,h}(v)w_{ij}p_{ij}}{\sum_{i=0}^{m}\sum_{j=0}^{n}N_{i,k}(u)N_{j,h}(v)w_{ij}} \tag{8.40}$$

có miền xác định là $[a,b]\times[c,d]$ và được gọi là bề mặt B-Spline hữu tỷ không đồng nhất (**N**on-**u**niform **r**ational **B**-**s**pline - NURBS).

Câu hỏi và bài tập

1. Đối với các đoạn cong, thế nào là liên tục bậc 0, 1 và 2 (C0, C1, C2)?
2. Hãy trình bày cách hiểu của bạn về các hàm cơ sở (basic function) của một đường cong tham số.
3. Nêu những điểm thuận lợi và bất lợi khi sử dụng đường cong NURBS?
4. Nêu sự khác nhau khi biểu diễn đường cong Hermite và đường B-spline?
5. **Bài tập lập trình:** Chương trình vẽ - hãy mở rộng chương trình đã phát triển trong bài tập lập trình ở chương trước để cho phép người sử dụng tạo ra một bề mặt B-spline bằng cách xác định lưới điểm điều khiển.

Chương 9
MÔ HÌNH ÁNH SÁNG TRONG ĐỒ HỌA MÁY TÍNH

Trong đồ họa máy tính, để có thể tạo ra các vật thể trông như thật, chúng phải tương tác với ánh sáng. Sự tương tác giữa vật liệu và ánh sáng này đòi hỏi phải có sự phát triển của các **mô hình sáng** (*illumination model*) cho sự truyền sáng, với mục tiêu là tạo ra độ chân thực ảnh. Các mô hình sáng là các luật đơn giản về tương tác giữa vật thể và ánh sáng. Nói một cách khác, chúng mô phỏng cách thức của các vật liệu được chiếu sáng, như là được quan sát trong thế giới thực. Mỗi một mô hình sáng đều cố gắng để các vật thể được kết xuất trong đồ họa máy tính trông tựa như các vật thể thật mà không phải thực hiện quá nhiều tính toán phức tạp. Tóm lại, các mô hình sáng tổng quát hóa và lý tưởng hóa cách thức tương tác giữa ánh sáng và các vật thể, với mỗi mô hình thể hiện một kiểu tương tác nhất định.

Sau khi đã xác định được các mặt hiện của các vật thể, chúng ta dùng **mô hình tạo bóng** (*shading model*) để thiết lập màu sắc và cường độ sáng tại tất cả các điểm trên các mặt đó. Những tính toán này thường tốn kém hơn hẳn quá trình xác định xem các mặt này có hiện hay không trừ trường hợp có rất nhiều vật thể.

Trước hết cần phân biệt mô hình sáng và mô hình tạo bóng. Khái niệm thứ hai toàn diện hơn khái niệm thứ nhất. Một mô hình tạo bóng sử dụng mô hình sáng. Hai mô hình tạo bóng khác nhau có thể sử dụng cùng một mô hình sáng. Ví dụ, một mô hình tạo bóng có thể tính cường độ ánh sáng tại mọi điểm theo mô hình sáng bất biến và một mô hình tạo bóng khác chỉ tính cường độ ánh sáng

tại các đỉnh của đa giác với cùng mô hình sáng và sau đó nội suy các giá trị này.

Có hai cách chọn mô hình sáng: một cách muốn mô phỏng một cách chính xác, và cách thứ hai chỉ cần đạt được độ chân thực nhất định. Cách thứ nhất đúng là hoàn hảo nhưng chắc chắn sẽ đòi hỏi rất nhiều tính toán. Cách thứ hai tính toán nhanh hơn nhiều và hi vọng vẫn tạo được những bức ảnh tốt. Tuy nhiên, không giống như lý thuyết, trên thực tế người ta không chia làm hai loại như vậy mà chia làm ba loại: **theo kinh nghiệm** (*empirical*), **truyền dẫn** (*transitional*), và **theo phân tích** (*analytical*). Ba loại kỹ thuật tạo bóng tương ứng là: **tăng dần** (*incremental*), **sử dụng dò tia** (*ray tracing*), và **sử dụng độ phát xạ** (*radiosity*). Ngoài ra người ta còn sử dụng phương pháp kết hợp của sử dụng dò tia và sử dụng radiosity.

Các mô hình sáng loại thứ nhất được phát triển dựa vào kinh nghiệm. Các giá trị độ sáng được tính toán sau khi các đối tượng hình học được biến đổi về không gian màn hình và các phương pháp tiếp cận đường quét tăng dần được sử dụng. Các mô hình truyền dẫn sử dụng quang học nhiều hơn. Chúng sử dụng nhiều hình học trong không gian vật thể do đó sự phản xạ, khúc xạ và bóng chính xác hơn về mặt hình học. Kỹ thuật dò tia bắt đầu từ đó. Dần dần, các mô hình phân tích được phát triển. Hai cách tiếp cận này được giải thích thông qua hai câu hỏi. Một cách tiếp cận bắt đầu từ mắt, quan tâm đến các mặt hiện, và đặt câu hỏi với mỗi điểm hiện:

"Cần thông tin gì để tính được mầu sắc của điểm mặt này?"

Để có được thông tin này, phải quan tâm đến các mặt khác nữa, và câu hỏi lại được hỏi một cách đệ quy. Rò tia là một ví dụ của cách tiếp cận này. Cách tiếp cận thứ hai bắt đầu từ các nguồn sáng, và đặt câu hỏi:

"Ánh sáng bị phản lại hay truyền qua mặt này như thế nào?"

Với cách tiếp cận này, mọi mặt được chiếu sáng đều trở thành vật phát sáng. Đây chính là cách sử dụng độ phát xạ hoạt động.

Hai hiện tượng quan trọng phải lưu ý khi mô phỏng ánh sáng là:

(1) Sự tương tác của ánh sáng với ranh giới của các vật liệu, và

(2) Sự phân tán và biến mất của ánh sáng khi đi qua các vật liệu.

Chính vì vậy, hai thành phần quan trọng của mọi mô hình sáng là các tính chất của bề mặt và tính chất của ánh sáng. Một tính chất quan trọng của một bề mặt là tính phản quang. Sự phản quang khác nhau với các bước sóng khác nhau tạo ra màu sắc. Một tính chất khác là độ trong suốt.

Có rất nhiều mô hình sáng mô phỏng sự ảnh hưởng trực tiếp của ánh sáng đến các vật thể - đó chính là các **mô hình sáng cục bộ** (*local illumination*). Một số mô hình sáng khác xử lý sự tương tác giữa ánh sáng và các vật thể một cách toàn cục – **mô hình sáng toàn cục** (*global illumination*).

9.1. Các mô hình sáng

9.1.1. Mô hình sáng cục bộ

Mô hình sáng cục bộ là mô hình sáng trực tiếp từ một nguồn sáng hữu hình. Mô hình này không quan tâm đến toàn bộ quá trình truyền ánh sáng mà chỉ quan tâm đến sự ảnh hưởng trực tiếp của ánh sáng đến các đối tượng khi được nhìn thấy trong một cảnh vật.

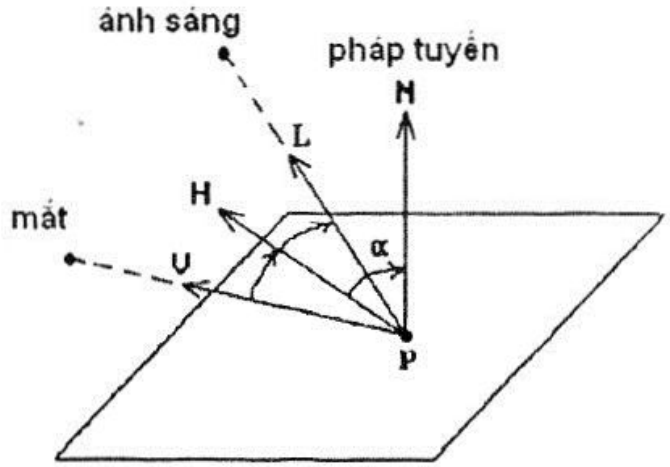
Mô hình sáng cục bộ bỏ qua sự phản quang giữa đối tượng và bỏ qua sự truyền ánh sáng trong môi trường. Mô hình sáng cục bộ là mô hình sáng trực tiếp cộng thêm sự kết hợp giữa ánh sáng môi trường với tính chất của vật liệu. Nói một cách khác, sự mô phỏng truyền ánh sáng trong mô hình sáng cục bộ chỉ là sự tính toán ánh sáng đi thẳng từ nguồn ánh sáng đến vật thể được chiếu và dừng ở đó. Rõ ràng đây là một mô hình sáng không đầy đủ.

Mô hình sáng cục bộ, dựa trên lý thuyết tia, coi ánh sáng phản quang có ba thành phần: thành phần **môi trường** (*ambient*), thành phần **khuyếch tán** (*diffuse*), và thành phần **phản chiếu** (*specular*).

Ánh sáng môi trường là một thiết lập ánh sáng có cường độ không đổi trong một cảnh vật, là tổng của tất cả các ánh sáng gián tiếp trong cảnh vật đó. Nói một cách khác, ánh sáng môi trường là

một kiểu mô hình tự phát sáng để bắt chước sự phản quang giữa vật thể. Ánh sáng môi trường có cường độ độc lập, không phụ thuộc vào các vật thể trong một cảnh vật. Thực tế, ánh sáng môi trường được đưa ra để cân bằng với ánh sáng trực tiếp thông qua việc giả lập sự phản quang giữa các vật thể sao cho bức ảnh tạo ra không bị tối và quá tương phản.

Thiết lập ánh sáng môi trường được tạo ra thông qua một thông số cho toàn cảnh vật hoặc riêng cho mỗi vật thể. Tuy nhiên, với hầu hết các cảnh vật, ánh sáng môi trường thường được đặt là 0 hoặc rất thấp trừ những vật thể tỏa sáng.



Hình 9.1. Một số ký hiệu để mô tả mô hình ánh sáng cục bộ.

Sau đây là một số ký hiệu chúng ta sẽ sử dụng để mô tả mô hình ánh sáng cục bộ (xem Hình 9.1). Tại mỗi điểm **p** của một bề mặt, véc-tơ pháp tuyến đơn vị của tiếp diện với bề mặt tại điểm đó được ký hiệu là **N**. Véc-tơ V và L là hai véc-tơ đơn vị trỏ đến mắt (hay máy quay) và nguồn sáng. Để thuận lợi, chúng ta định nghĩa một véc-tơ đơn vị H là đường phân giác của góc tạo bởi V và L

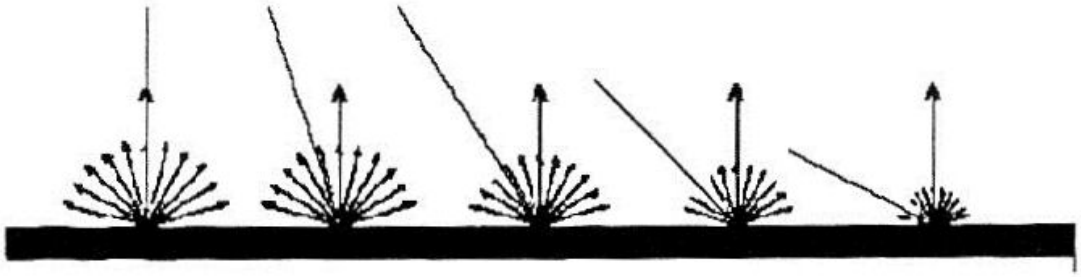
$$H = \frac{V + L}{|V + L|}.$$

Góc giữa H và N được gọi là α.

Mô hình đơn giản nhất chỉ quan tâm đến ánh sáng môi trường và ánh sáng khuếch tán. Thành phần ánh sáng sáng môi trường có dạng:

$$I_a(\lambda)k_a(\lambda),$$

với $I_a(\lambda)$ là cường độ ánh sáng môi trường với bước sóng λ và $k_a(\lambda) \in [0,1]$ là hệ số phản quang ánh sáng môi trường, là lượng ánh sáng môi trường được phản quang lại.



Hình 9.2. Ánh sáng khuếch tán theo luật Lambert.

Thành phần khuyếch tán có dạng

$$I_p(\lambda)k_d(\lambda)r_d,$$

với $I_p(\lambda)$ là cường độ của nguồn sáng tới điểm **p**, $k_d(\lambda) \in [0,1]$ là hệ số khuyếch tán, là một hằng số phụ thuộc vào vật liệu, và $\mathbf{r_d}$ là tỉ lệ khuyếch tán. Tỉ lệ $\mathbf{r_d}$ được tính từ luật Lambert cho khuyếch tán hoàn hảo, có nghĩa là ánh sáng khuyếch tán như nhau theo mọi hướng (xem Hình 9.2). Một vùng $\mathbf{A_1}$ ánh sáng đến theo hướng **L** sẽ chiếu trong khu vực $\mathbf{A_2}$ của mặt phẳng với véc-tơ pháp tuyến **N**. Nếu **θ** là góc giữa **N** và **L**, dễ dàng chứng minh rằng:

$$\frac{A_1}{A_2} = |\cos\theta| = |N \bullet L|.$$

Tỉ lệ $\mathbf{A_1/A_2}$ chỉ ra phần nào của ánh sáng sẽ khuyếch tán đều theo mọi hướng, tuy nhiên với một giả thiết nữa. Ánh sáng đến từ phía sau bề mặt sẽ không tham gia vào ánh sáng khuyếch tán. Điều đó có nghĩa là nếu bề mặt quay đi khỏi ánh sáng, khi $\mathbf{N \bullet L < 0}$, không có đóng góp gì. Như vậy $\mathbf{r_d = max(N \bullet L, 0)}$.

Tổng hợp lại, chúng ta có mô hình sáng Bouknight với ánh sáng môi trường và ánh sáng khuyếch tán:

$$I(\lambda) = I_a k_a(\lambda) + I_p(\lambda) k_d(\lambda) r_d.$$

Tiếp theo, chúng ta muốn có thành phần phản chiếu trong hàm cường độ của chúng ta. Cho một tia sáng L và véc-tơ pháp tuyến N của một mặt phẳng, công thức cho hướng phản chiếu R theo gương khi chạm vào mặt phẳng là

$$R = L - 2(L - (N \square L)N).$$

Người ta đã giả thiết thành phần phản chiếu có dạng

$$I_p(\lambda) k_s r_s,$$

với $\mathbf{k_s}$ là hệ số phản chiếu và $\mathbf{r_s}$ - tỉ lệ phản chiếu, là một hàm của góc $\boldsymbol{\theta}$, nhưng thường được đặt từ **0** đến **1**. Trong trường hợp phản chiếu hoàn hảo

$$r_s = \begin{cases} 1 & \text{nếu } V=R \\ 0 & \text{nếu ngược lại.} \end{cases}$$

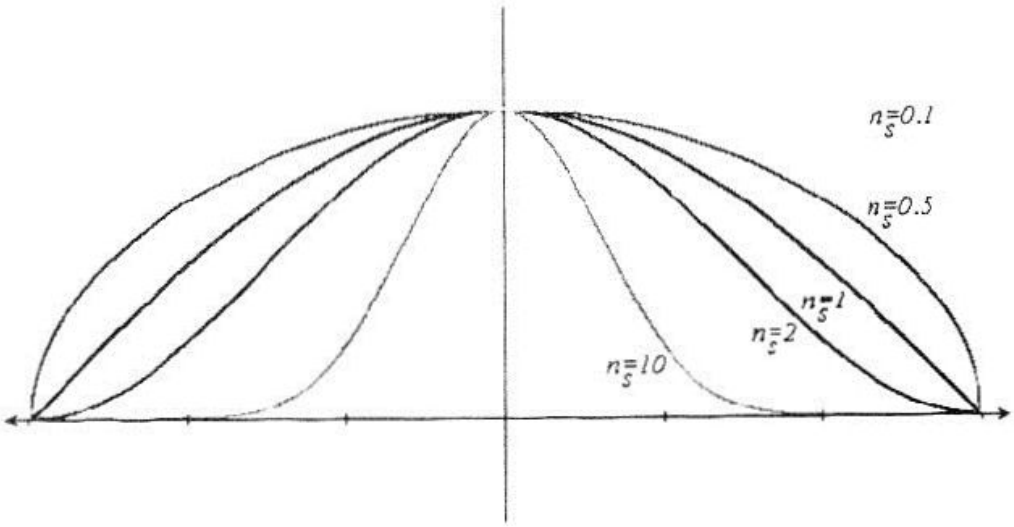
Tuy nhiên, điều này không thực tế. Có thể thấy rằng có sự phản chiếu khi **V** gần **R**. Phong đã dùng góc $\boldsymbol{\varphi}$ giữa **R** và **V** để điều khiển sự suy giảm của cường độ ánh sáng phản chiếu khi rời khỏi hướng phản chiếu gương. Cụ thể, ông dùng lũy thừa bậc $\mathbf{n_s}$ của $\mathbf{\cos \varphi = R \bullet V}$ để điều chỉnh độ sắc nét của điểm phản chiếu và định nghĩa $\mathbf{r_s}$ bởi:

$$r_s = r_s(\phi) = \max(\cos^{n_s}\phi, 0) = \max((R \square V)^{n_s}, 0)$$

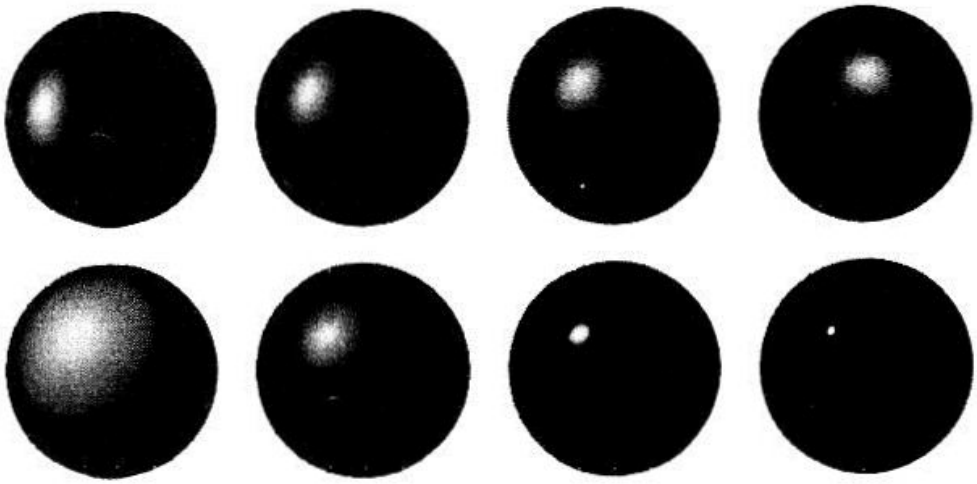
Mô hình phản chiếu của Phong:

$$I(\lambda) = I_a(\lambda) k_a(\lambda) + I_p(\lambda)(k_d(\lambda) r_d + k_s r_s)$$

Sự thay đổi của n_s tạo nên sự khác biệt biệt về kích thước của **điểm phản chiếu** (*highlight*) (xem Hình 9.3 và Hình 9.4). Giá trị thường dùng của $\mathbf{n_s}$ là từ **50** đến **60**. Lưu ý rằng, hệ số phản chiếu $\mathbf{k_s}$ không phải là một hàm của bước sóng. Như vậy, điểm phản chiếu có màu của nguồn sáng.



Hình 9.3. Tác dụng của n_s đối với kích thước của điểm phản chiếu.



Hình 9.4. Ví dụ về kích thước của điểm phản chiếu với các hệ số n_s khác nhau.

Như vậy mô hình sáng cục bộ chỉ là giải pháp cho câu hỏi ánh sáng ảnh hưởng trực tiếp đến các đối tượng được chiếu sáng như thế nào. Câu hỏi tiếp theo được đặt ra là ánh sáng tương tác với các vật hữu hình trong một cảnh vật như thế nào? Câu hỏi này được giải quyết bằng các mô hình sáng toàn cục.

9.1.2. Mô hình sáng toàn cục

Mô hình sáng toàn cục là mô hình truyền ánh sáng đầy đủ hơn. Mô hình sáng toàn cục tính đến sự truyền ánh sáng gián tiếp có phản quang trong một cảnh vật.

Có hai loại cài đặt mô hình sáng toàn cục. Một loại mô phỏng cách thức của **ánh sáng phản chiếu hoàn hảo** (*perfect specular light*); loại khác mô phỏng hoạt động của **ánh sáng khuyếch tán hoàn hảo** (*perfect diffuse light*). Loại cài đặt đầu tiên thường được gọi là phản chiếu lý tưởng – cách mà kỹ thuật dò tia làm. Loại thứ hai, khuyếch tán hoàn hảo, hay còn được gọi là **khuyếch tán lý tưởng** (*ideal diffuse reflection*), thường được tính toán sử dụng kỹ thuật độ phát xạ.

Mô hình sáng toàn cục thực sự là lĩnh vực nghiên cứu mô phỏng sự truyền ánh sáng áp dụng vào đồ họa máy tính. Mô hình sáng toàn cục ban đầu là một kỹ thuật để tạo nên các bức ảnh đồ họa máy tính thực tế hơn và dần dần tiến đến là bắt chước sự truyền ánh sáng. Về bản chất, mô hình sáng toàn cục là cách thể hiện được toàn bộ sự truyền ánh sáng trong một cảnh vật cho trước. Dù là ánh sáng đến trực tiếp từ nguồn sáng hay phản quang xung quanh môi trường, ánh sáng toàn cục sẽ tính toán và quan tâm đến nó.

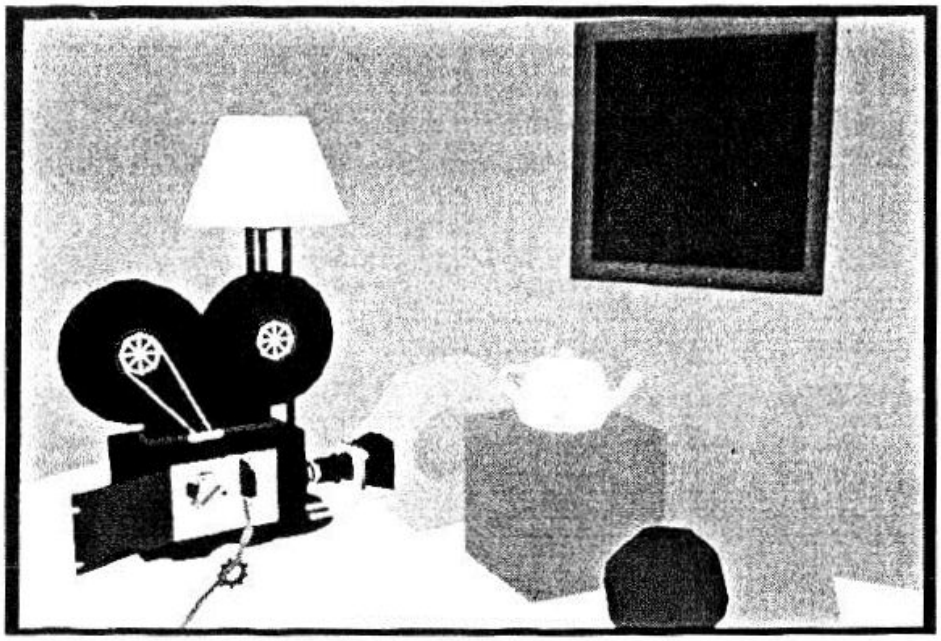
Để có một giải pháp hoàn thiện, người ta có thể kết hợp kỹ thuật dò tia với kỹ thuật độ phát xạ. Đây được gọi là **giải pháp hai-pha** (*two-pass solution*). Độ phát xạ được tính trước, và sau đó các đường dò tia được tính để tính ánh sáng trực tiếp cũng như ánh sáng phản xạ và ánh sáng khuyếch tán.

9.2. Các mô hình tạo bóng

Để có thể có được một mô hình truyền ánh sáng hoàn chỉnh, không chỉ cần mô phỏng ánh sáng trực tiếp và sự phản quang, mà còn phải mô phỏng cách mà ánh sáng ảnh hưởng đến vật liệu của các đối tượng. Đây được gọi là các mô hình tạo bóng.

9.2.1. Tạo bóng bất biến

Tạo bóng bất biến (*constant shading*) là cách gán một màu cho toàn bộ vật thể. Đây thực sự không sử dụng cơ chế tạo bóng nào và là mô hình tạo bóng sơ khai nhất.

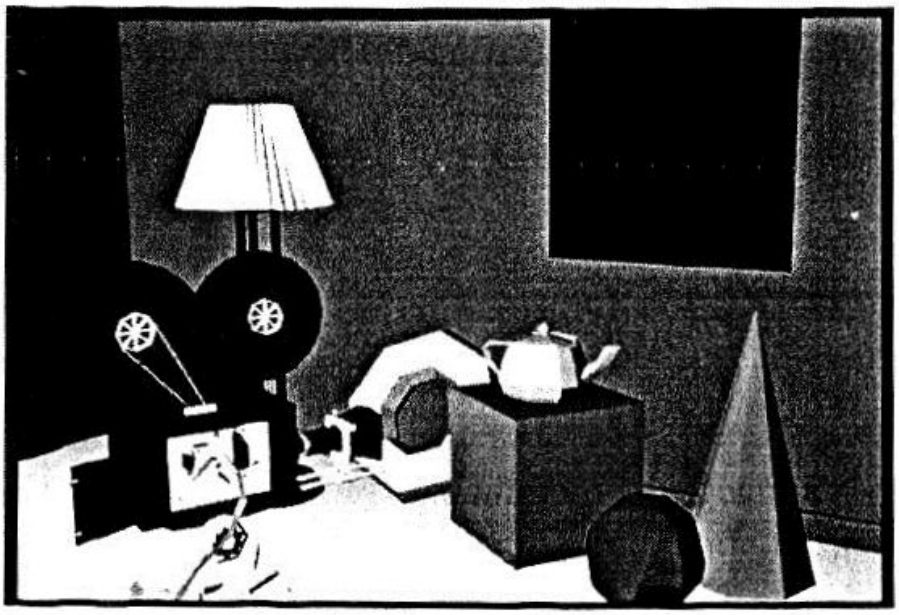


Hình 9.5. Tạo bóng bất biến.

9.2.2. Tạo bóng phẳng

Tạo bóng phẳng (*flat shading*) có liên quan đến mô hình tạo bóng bất biến tuy nhiên hình thức và cách thức thì hơi khác. Đây chính là cách tạo bóng bất biến cho mỗi đa giác. Tạo bóng phẳng tạo ra cho các vật thể có cái nhìn khối và ba chiều.

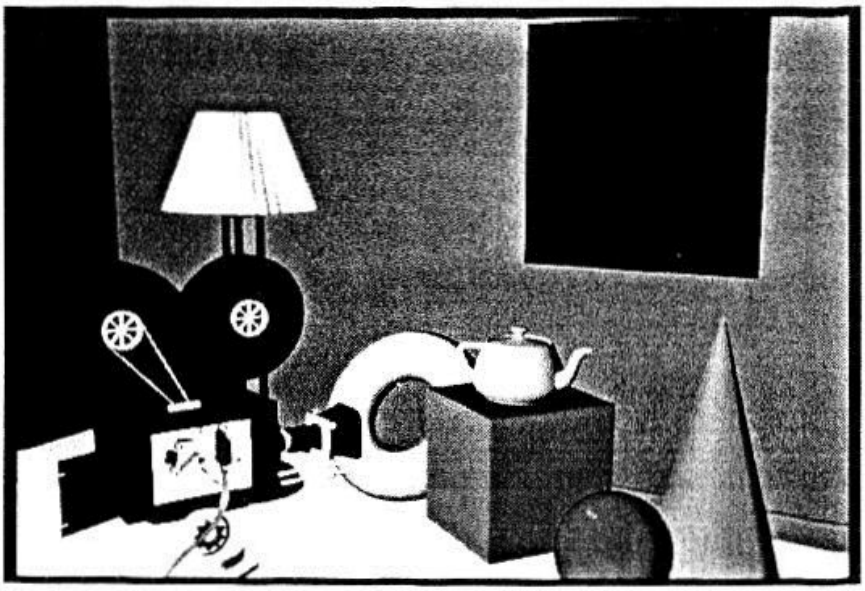
Tạo bóng phẳng có tính đến ánh sáng môi trường cùng với ánh sáng khuyếch tán. Mô hình này tính toán rất nhanh, tuy nhiên các vật thể trông giả tạo và góc cạnh.



Hình 9.6. Tạo bóng phẳng.

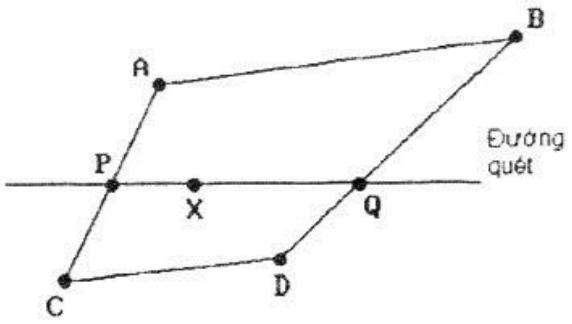
9.2.3. Tạo bóng Gouraud

Tạo bóng Gouraud (*Gouraud shading*), được đặt tên theo nhà khoa học Hendri Gouraud, là sự mô phỏng các bề mặt nhẵn nhụi và mờ. Tên kỹ thuật của tạo bóng Gouraud là **tạo bóng nội suy cường độ** (*intensity interpolation shading*), hay tạo bóng nội suy màu sắc bởi vì nó tính cường độ của mỗi đỉnh đa giác và sau đó nội suy ra toàn bộ đa giác. Bằng cách thực hiện nội suy cường độ này, nó loại bỏ sự hiện diện của biên giới giữa các đa giác, làm cho chúng trông nhẵn nhụi.



Hình 9.7. Tạo bóng Gouraud.

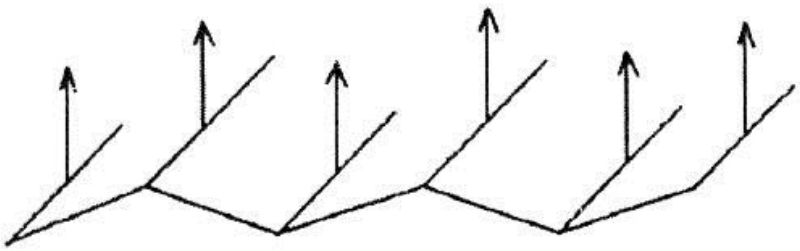
Thuật toán tạo bóng Gouraud tính độ sáng tại mỗi đỉnh của một vật thể với một mô hình sáng thích hợp rồi sau đó tính ra độ sáng tại mọi điểm bằng nội suy theo các đường quét. Hình 9.8 cho thấy một ví dụ. Giả sử, độ sáng đã biết tại các đỉnh **A**, **B**, **C**, và **D**, người ta tính độ sáng tại điểm **X** như sau: gọi **P** và **Q** là điểm mà cạnh **AC** và **BD** cắt đường quét chứa **X**. Tính độ sáng tại điểm **P** và **Q** bằng nội suy tuyến tính các giá trị độ sáng tại **A**, **C**, và **B**, **D**. Độ sáng tại **X** là nội suy tuyến tính của độ sáng tại **P** và **Q**.



Hình 9.8. Ví dụ về tính toán trong thuật toán tạo bóng Gouraud.

Để có được giá trị độ sáng tại các đỉnh đa giác, chúng ta cần tính véc-tơ pháp tuyến. Giá trị này thường được tính bằng cách lấy trung bình véc-tơ pháp tuyến của các mặt kề với đỉnh đó. Lưu ý rằng, cách lấy trung bình này tạo ra hiệu ứng trơn cho vật thể. Đây là điều chúng ta muốn khi ta xấp xỉ một vật thể cong bằng các mặt đa giác. Tuy nhiên, vì cường độ được trải đều trên toàn đa giác, nó không thể cho thấy rõ một số điểm phản chiếu. Thay vào đó, tạo bóng Gouraud trải đều điểm phản chiếu ra đều đa giác.

Trong một số trường hợp khác, đôi khi chúng ta muốn thể hiện rõ các cạnh, ví dụ như một hình lập phương, chúng ta phải tìm cách tránh hiệu ứng trơn này. Điều này có thể thực hiện được bằng cách tô bóng các điểm của một mặt sử dụng véc-tơ pháp tuyến là véc-tơ pháp tuyến của mặt đó.



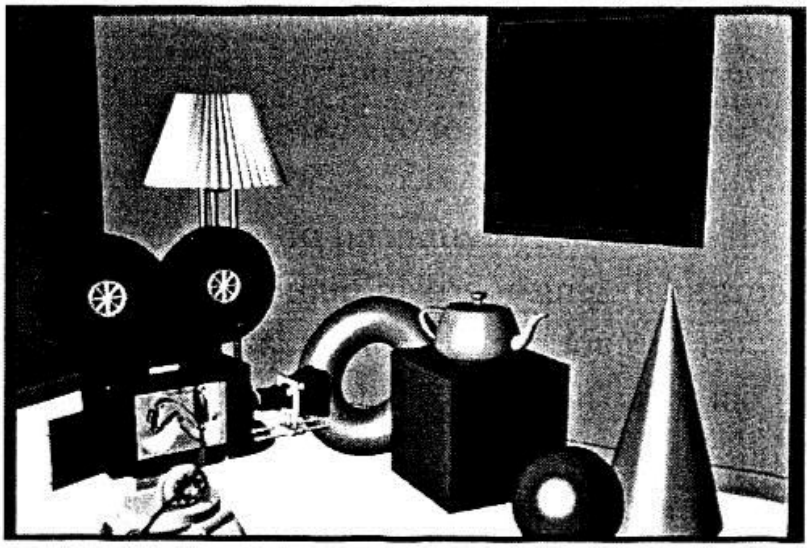
Hình 9.9. Vấn đề của tạo bóng nội suy.

Một số vấn đề có thể xảy ra với tạo bóng Gouraud, hay chính xác hơn là của tạo bóng nội suy. Hình 9.9 cho thấy các bề mặt được dùng để xấp xỉ một bề mặt sóng, tuy nhiên, với tạo bóng nội suy, chúng sẽ trông như là phẳng vì cách lấy trung bình các véc-tơ pháp tuyến. Cách đơn giản nhất để giải quyết vấn đề này là chia nhỏ các mặt ra nữa.

Ngoài ra còn có một vấn đề nữa với tạo bóng Gouraud. Nếu sử dụng mô hình sáng của Phong, một sự thay đổi nhỏ ở véc-tơ pháp tuyến sẽ kéo theo sự thay đổi lớn ở phần ánh sáng phản chiếu. Vấn đề này sẽ tạo ra những điểm phản chiếu bất thường trong tạo bóng Gouraud.

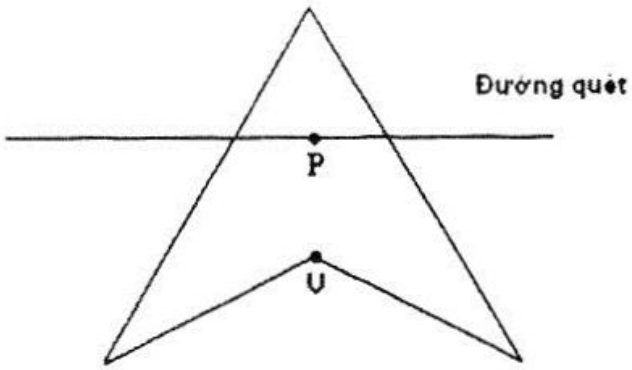
9.2.4. Tạo bóng Phong

Tạo bóng Phong (*Phong shading*), được đặt theo tên nhà khoa học Bùi Tường Phong, là sự mô phỏng các bề mặt bóng. Nó cũng được gọi là tạo bóng nội suy véc-tơ pháp tuyến. Nó có liên quan đến tạo bóng Gouraud nhưng xử lý các điểm phản chiếu tốt hơn bằng cách nội suy các véc-tơ pháp tuyến của bề mặt thay vì nội suy cường độ. Việc này được thực hiện với từng điểm trong quá trình kết xuất. Bằng cách nội suy các véc-tơ hướng của bề mặt (véc-tơ pháp tuyến) để mô phỏng sự phản chiếu ánh sáng, tạo bóng Phong có khả năng mô phỏng các điểm phản chiếu tốt hơn tạo bóng Gouroud. Tạo bóng Phong tính đến vị trí của người quan sát, hướng của bề mặt, và hướng phản xạ ánh sáng. Sử dụng tạo bóng Phong, ta có thể thay đổi kích thước của các điểm phản chiếu, tuy nhiên không thay đổi được màu sắc của điểm phản chiếu. Màu sắc của điểm phản chiếu trong tạo bóng Phong phụ thuộc vào màu sắc của ánh sáng có mặt trong cảnh. Tạo bóng Phong có thể được coi như là sự mô phỏng không đầy đủ của sự phản chiếu ánh sáng. Nó chỉ quan tâm đến cường độ và vị trí của nguồn sáng.



Hình 9.10. Tạo bóng Phong.

Tạo bóng Phong cho kết quả tốt hơn tạo bóng Gouraud, tuy nhiên nó cũng có những vấn đề của nó. Xét đa giác lõm trong Hình 9.11 Sự khác biệt giữa véc-tơ pháp tuyến nội suy tại điểm **P** và véc-tơ pháp tuyến tại điểm **V** có thể tạo nên một sự thay đổi lớn về cường độ sáng từ **P** sang **V**.



Hình 9.11. Vấn đề của tạo bóng Phong

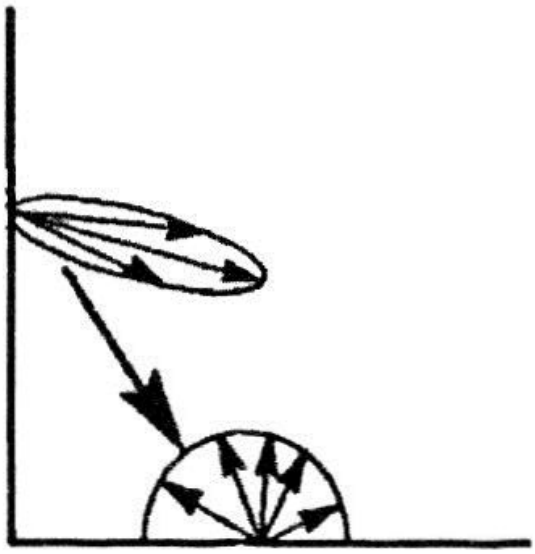
9.2.5. Tạo bóng Lambertian

Tạo bóng Lambertian (*Lambertian shading*), được đặt tên theo nhà khoa học Johann Heinrich Lambert, là sự mô phỏng các bề mặt mờ và xỉn. Đây cũng được gọi là mô hình tạo bóng khuyếch tán lý tưởng, hay tạo bóng cosin. Tạo bóng Lambertian là mô hình ánh sáng tiến vào một vật thể (ánh sáng bất thường) mà có sự phản xạ ra không đổi không phụ thuộc vào góc nhìn của người quan sát. Nói một cách khác, tạo bóng Lambertian giả thiết rằng ánh sáng đến phản xạ đều theo mọi hướng. Góc của ánh sáng đến không ảnh hưởng các hướng ánh sáng phản xạ ra.

Một cách trực quan, tạo bóng Lambertian mô phỏng bề ngoài của các vật liệu như viên phấn, bột hay bìa giấy. Tuy nhiên, trên thực tế không tồn tại những vật liệu kiểu Lambertian, vì chúng thường là sự kết hợp của khuyếch tán, phản xạ và khuyếch tán có hướng. Bằng việc gán một bề mặt là Lambertian, chúng ta đã đơn giản hóa quá trình tính toán vì chúng ta không phải quan tâm đến hướng của ánh sáng đi đến và đi ra và ánh sáng thay đổi bề mặt vật thể như thế nào. Chính vì thế hầu hết các hệ thống kết xuất sử dụng độ phát xạ thường giả thiết các bề mặt vật thể trong cảnh là Lambertian.

9.2.6. Tạo bóng Blinn

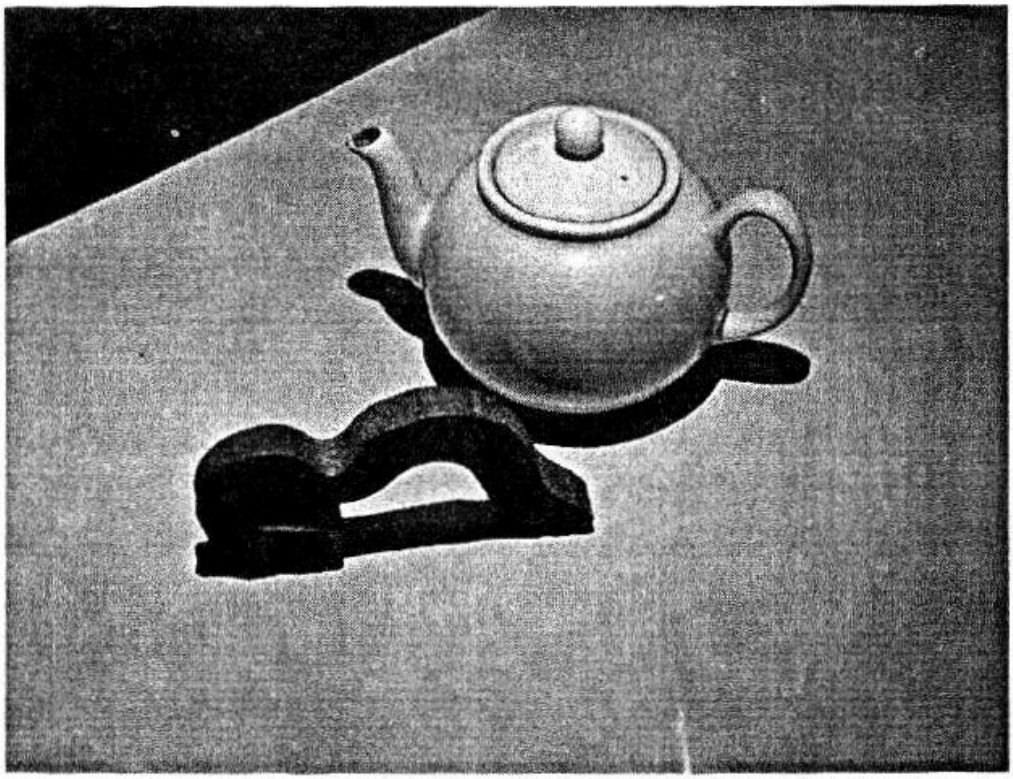
Tạo bóng Blinn (*Blinn shading*), được đặt tên theo James Blinn, là sự áp dụng trong đồ họa máy tính của mô hình tạo bóng Torrance-Sparrow-Cook (theo tên 3 nhà khoa học Kenneth E. Torrance, Ephraim M. Sparrow và Robert L. Cook), dựa trên các loại phản xạ **"từ phản chiếu đến khuyếch tán"** (*specular-to-diffuse*) (xem Hình 9.12). Mô hình Torrance-Sparrow-Cook giả thiết rằng bề mặt của vật thể được tạo ra bởi các mặt rất nhỏ tự tạo bóng. Nó cũng tính đến các điểm sáng trên các cạnh của một số vật liệu khi được quan sát từ những góc nhất định. Đây là một mô hình tạo bóng dựa trên vật lý. Mô hình Torrance-Sparrow-Cook được James Blinn áp dụng đầu tiên vào đồ họa máy tính; chính vì thế được gọi là tạo bóng Blinn.



Hình 9.12. Loại phản xạ "từ phản chiếu đến khuyếch tán".

9.2.7. Dò tia

Dò tia (*ray tracing*) là kỹ thuật ánh sáng toàn cục mà sử dụng các tia hay các phô-tông để lưu vết những đường sáng trong một cảnh vật khi nó được chiếu lên mặt phẳng quan sát 2D (màn hình). Chính vì thế dò tia ngược từ mắt phụ thuộc vào khung nhìn. Có nghĩa là, nó chỉ tính toán những bề mặt hữu hình từ góc nhìn của người quan sát. Nó bỏ qua những bề mặt mà người quan sát không nhìn thấy. Đây chính là kỹ thuật dò tia ngược cổ điển.



Hình 9.13. Kết quả của kỹ thuật dò tia.

Dò tia ngược bắt đầu với việc thiết lập một mặt phẳng quan sát 2D, được chia thành một lưới tương đối mịn. Lưới này là đại diện của thiết bị hiển thị (màn hình). Mỗi ô vuông trên lưới thể hiện một thành phần điểm, hay còn gọi là pixel. Nhiệm vụ của bộ dò là để xác định màu và độ sáng của mỗi điểm dựa trên các thông tin hiện có về cảnh. Đây chính là câu hỏi về sự hữu hình của các vật thể trong một cảnh vật đối với người quan sát.

Cách làm thông thường là "bắn" một tia từ gốc của điểm ảnh và ngược vào cảnh cho đến khi nó va vào vật gì đó, lướt qua nó, hoặc biến mất trong cảnh. Khi tia va vào hoặc lướt qua một vật gì đó có nghĩa là có một vật thể ở đó – một thực tế quan trọng liên quan đến độ sâu. Câu hỏi về sự hữu hình đã trở thành câu hỏi về sự che lẫn nhau: Các vật thể nào ở gần nhất mà lại hữu hình trong cảnh vật?

Khi tia gặp một bề mặt, nó phải xác định xem bề mặt đó có tính phản xạ, khúc xạ hay phát sáng. Các bề mặt phản xạ thì phản xạ ánh sáng lại, các bề mặt khúc xạ chuyển hướng ánh sáng, và các bề mặt phát sáng là các nguồn sáng. Mọi tia phải

"hỏi" câu hỏi này khi nó đi ngược từ mắt tiến đến một bề mặt. Đây được gọi là kỹ thuật "dò tia ngược" hay "dò tia ánh sáng" được định nghĩa bởi James Arvo. Đây chính là lý do mà dò tia đòi hỏi rất nhiều tính toán.

Khi mà tia đã biết được loại của bề mặt mà nó sắp tiếp xúc, nó phải tính toán xem nó sẽ gặp bề mặt kia như thế nào. Điều này được thực hiện bằng cách tạo ra một tia mới từ điểm (spot). Tia này bắn ra những mảnh nhỏ từ điểm đó để tính được gốc của tia giao này. Nếu một số mảnh nhỏ không chạm vào vật thể, chúng có thể được bỏ qua vì sẽ không có ánh sáng đến từ hướng đó. Nếu những mảnh nhỏ đậu vào vật thể mới, những tia mới được tạo ra từ điểm đó. Quá trình này được lặp lại cho đến khi toàn bộ nguồn của ánh sáng đã được biết. Với mỗi tia mới được tạo ra, nguồn của nó phải được truy vết và biết cho đến khi toàn bộ ánh sáng đã được tính đến. Quá trình truy vết này được thực hiện với mọi điểm trên lưới quan sát 2D. Chính vì thế, nếu độ phân giải càng cao, thì càng cần tính toán nhiều do phải dùng nhiều tia hơn.

Để xác định các bóng, các **tia bóng** (*shadow rays*) được sử dụng. Đây là những tia phụ thêm được bắn ra từ điểm đến mỗi nguồn sáng. Nếu tia bị chắn trước khi đến được nguồn sáng, có nghĩa là không có ánh sáng đến từ hướng đó. Ngược lại, nếu tia bóng đến được nguồn sáng mà không vướng phải chướng ngại vật nào, thì điểm đó thể hiện nguồn sáng.

Tính chất đệ qui của dò tia đã biến nó thành kỹ thuật ánh sáng toàn cục. Kỹ thuật dò tia ngược từ mắt nhìn đã được biết đến là dò tia trong đồ họa máy tính.

Cũng có thể dò tia từ nguồn sáng và xem chúng đến được mắt người quan sát như thế nào. Quá trình này gọi là dò tia tiến. Dò tia ngược được sử dụng rộng rãi hơn vì nó hiệu quả hơn dò tia xuôi. Dò ánh sáng từ nguồn tưởng như là cách tốt nhất để làm "dò tia"; nhưng thực ra, hầu hết các tia sáng từ một nguồn sáng có đóng góp rất nhỏ vào việc tạo ra bức ảnh. Như Andrew Glassner đã nói: "Một lượng rất nhỏ phô-tôn rời mặt trời để đóng góp vào cảnh đẹp của công viên Grand Canyon".

Dò tia ngược và tiến có thể được kết hợp để tạo ra các hiệu ứng ánh sáng thực tế, nhất là trong các mô phỏng tập trung vào các điểm phản chiếu. Quá trình này được gọi là dò tia hai hướng, từ cả mắt lẫn nguồn sáng. Hầu hết các bộ dò tia đều không phải là hai hướng.

Mặc dù dò tia ngược cổ điển là một quá trình ánh sáng toàn cục mà tính toán ba loại phản chiếu (phản xạ, khúc xạ, và khuyếch tán), nhưng chỉ có thành phần phản xạ được dò ngược. Do đó, dò tia chỉ là một giải pháp ánh sáng phản xạ toàn cục.

9.3. Bài đọc thêm

Bùi Tường Phong (1942-1975) là một nhà nghiên cứu tiên phong trong lĩnh vực đồ họa máy tính. Các công trình nghiên cứu của ông thường được tham khảo đến bằng họ của ông, Bùi, tuy nhiên các sáng chế của ông lại được nhớ đến thông qua tên ông, Phong.

Bùi Tường Phong sinh ngày 14 tháng 12 năm 1942 tại Hà Nội. Sau khi học tại trường Lycee Albert Sarraut ở Hà Nội, ông chuyển vào Nam cùng gia đình năm 1954. Ở đó ông đã học trường Lycee Jean Jacques Rousseau. Ông đến Pháp năm 1964 và được nhận vào trường Ecole d'Ingenieur de Grenoble (ENSEHRMAG). Ông nhận học vị Licence es Science từ Grenoble năm 1966 và học vị Diplome d'Ingenieur từ ENSEEIHT, Toulouse, vào năm 1968. Ông tham gia Institut de Recherche d'Ingenieur et d'Automatique (IRIA) vào năm 1968 với tư cách là một nghiên cứu viên trong lĩnh vực Khoa học máy tính. Ở đó, ông đã tham gia phát triển hệ điều hành cho các máy tính số. Ông đến trường đại học Utah vào tháng 9 năm 1971 với tư cách là trợ lý nghiên cứu trong lĩnh vực Khoa học máy tính. Một bi kịch của cuộc đời ông chính là ông đã biết mình bị bệnh nặng từ khi còn là sinh viên. Sau khi rời trường đại học Utah, ông đến trường Stanford làm giáo sư và mất ngay sau đó vì bệnh ung thư máu.

Theo như giáo sư Ivan Sutherland, một người bạn của ông Phong và cũng rất nổi tiếng trong lĩnh vực đồ họa máy tính, ông Phong là một người rất thông minh và lịch lãm. Giáo sư Sutherland nhắc lại lời của ông Phong trong việc sinh ra cảnh vật trong đồ họa máy tính: "Tôi không mong muốn thể hiện mọi vật một cách

chính xác như chính nó trong thế giới thật, tôi chỉ mong muốn thể hiện nó một cách gần giống với một độ chính xác chấp nhận được mà thôi."

Bùi Tường Phong là người phát minh ra mô hình phản quang Phong và phương pháp tạo bóng nội suy Phong, là những kỹ thuật được dùng rộng rãi trong đồ họa máy tính. Ông công bố những thuật toán này trong luận án tiến sỹ của ông năm 1973 và một bài báo vào năm 1975. Ông nhận học vị tiến sỹ từ trường đại học Utah vào năm 1973.

Câu hỏi và bài tập

1. Hãy nêu ưu điểm và nhược điểm của phương pháp dò tia.
2. Hãy nêu nhược điểm của các phương pháp tạo bóng nội suy.
3. Tạo bóng Phong khắc phục được nhược điểm gì của tạo bóng Gouraud?
4. **Bài tập lập trình:** Chương trình vẽ - hãy mở rộng chương trình đã phát triển trong bài tập lập trình ở chương trước để cho phép người sử dụng đặt chất liệu bề mặt cho các mặt tam giác và cho phép người sử dụng đặt các nguồn sáng vào cảnh vật trong chương trình vẽ. Sau đó, cho phép cảnh vật được hiển thị với ánh sáng.

Chương 10
GIỚI THIỆU OPENGL

OpenGL là một giao diện phần mềm cho phần cứng đồ họa. Giao diện này gồm khoảng 150 lệnh khác nhau để xác định những đối tượng và thao tác để những ứng dụng tương tác ba chiều. OpenGL được thiết kế như một giao diện độc lập với phần cứng và xử lý theo luồng nối tiếp để triển khai trên nhiều nền tảng phần cứng khác nhau. Để thu được đặc tính đó, OpenGL không bao gồm những lệnh để thực hiện những tác vụ cửa sổ. OpenGL cũng không cung cấp những lệnh mức cao cho vẽ những đối tượng ba chiều. Chẳng hạn, những câu lệnh OpenGL không cho phép chúng ta trực tiếp thể hiện những hình phức tạp như xe ô tô, những phần của cơ thể, máy bay hoặc những phân tử hóa học. Với OpenGL, chúng ta cần dựng lên những mô hình mong muốn từ một tập nhỏ những hình học cơ bản như điểm, đường thẳng và đa giác.

Người ta có thể xây dựng thêm thư viện phụ trợ trên OpenGL để cung cấp những lệnh mức cao hơn. Thư viện tiện ích OpenGL (GLU, OpenGL Unitity Library) cung cấp những đặc tính được mô hình hóa, chẳng hạn mặt bậc hai, đường cong và mặt phức tạp. GLU là một phần chuẩn của mỗi bổ sung OpenGL.

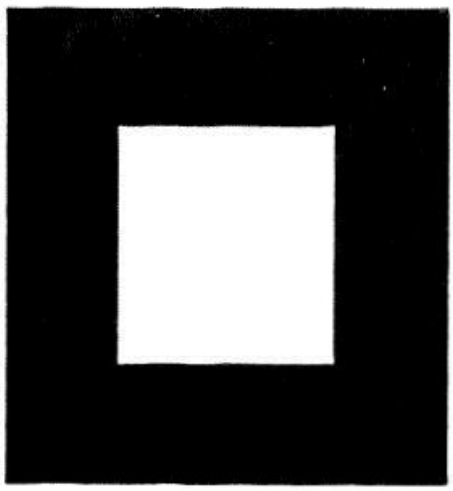
OpenGL không cung cấp giao diện lập trình để thể hiện hay biểu diễn các đối tượng phức tạp. Nó chỉ cung cấp giao diện thể hiện các hình học nguyên thủy, hoặc sắp xếp những đối tượng trong không gian ba chiều và chọn điểm nhìn thuận lợi. Nó có thể tính toán màu của tất cả đối tượng. Màu đó được xác từ những điều kiện ánh sách xác định, hoặc thu được từ việc dán một ảnh chất liệu lên đối tượng. Đồng thời, OpenGL biến đặc tả toán học của những đối tượng và những thông tin màu sắc liên quan để vẽ chúng trên màn hình.

Chương này giới thiệu một số khái niệm cơ bản trong OpenGL, với mục tiêu cung cấp cho người đọc một cái nhìn tổng quan về OpenGL. Các nội dung trong chương được trình bày để người đọc nắm được cấu trúc của một chương trình OpenGL và có thể cài đặt một chương trình OpenGL đơn giản. Các khái niệm nâng cao và chuyên sâu của OpenGL có thể được tra cứu thông qua các tài liệu kỹ thuật của OpenGL.

10.1. Một chương trình OpenGL đơn giản

Bởi vì chúng ta có thể làm rất nhiều điều với hệ thống đồ họa OpenGL, một chương trình OpenGL có thể rất phức tạp. Tuy nhiên, cấu trúc cơ bản của một chương trình OpenGL hữu dụng có thể đơn giản: Nhiệm vụ của nó là khởi tạo những trạng thái điều khiển việc OpenGL kết xuất thế nào và mô tả các đối tượng được kết xuất thế nào.

Ví dụ 10.1 cho thấy một chương trình OpenGL cơ bản với nhiệm vụ vẽ một đa giác màu trắng như trong Hình 10.1.



Hình 10.1. Kết quả của chương trình trong Ví dụ 10.1.

Ví dụ 10.1

```
#include <GL/gl.h>
main()
{
      InitializeAWindowPlease();
      glClearColor (0.0, 0.0, 0.0, 0.0);
```

```
    glClear (GL_COLOR_BUFFER_BIT);
    glColor3f (1.0, 1.0, 1.0);
    glOrtho(0.0, 1.0, 0.0, 1.0, -1.0, 1.0);
    glBegin(GL_POLYGON);
    glVertex3f (0.25, 0.25, 0.0);
    glVertex3f (0.75, 0.25, 0.0);
    glVertex3f (0.75, 0.75, 0.0);
    glVertex3f (0.25, 0.75, 0.0);
    glEnd();
    glFlush();
    UpdateTheWindowAndCheckForEvents();
}
```

Trong **Ví dụ 10.1**, dòng đầu tiên của thủ tục **main()** khởi tạo một cửa sổ trên màn hình: Thủ tục **InitializeAWindowPlease()** là nơi để đặt các thủ tục khởi tạo cửa sổ - đây không phải là một thủ tục của OpenGL. Hai dòng tiếp theo là hai lệnh OpenGL để xóa cửa sổ về màu đen **glClearColor()** thiết lập màu để xóa cửa sổ, và **glClear()** thực hiện xóa cửa sổ. Sau khi màu xóa cửa sổ được thiết lập, cửa sổ sẽ được xóa với màu đó mỗi khi **glClear()** được gọi. Màu xóa này sẽ được thay đổi khi **glClearColor()** lại được gọi. Tương tự như vậy, lệnh **glColor3f()** thiết lập màu được dùng để vẽ các vật thể - trong trường hợp này là màu trắng. Tất cả các vật thể sẽ được vẽ bằng màu này cho đến khi lệnh thiết lập màu vẽ khác được gọi.

Lệnh OpenGL tiếp theo trong chương trình, **glOrtho()**, xác định hệ tọa độ mà OpenGL sẽ dùng để vẽ bức ảnh cuối cùng và bức ảnh đó được thể hiện lên màn hình thế nào. Các lệnh tiếp theo, được đặt trong dấu ngoặc giữa **glBegin()** và **glEnd()**, định nghĩa vật thể sẽ được vẽ - trong ví dụ này là một đa giác có bốn đỉnh. Các đỉnh của đa giác được định nghĩa bởi lệnh **glVertex3f()**. Như chúng ta có thể đoán từ tham số của lệnh, chính là tọa độ (x, y, z), đa giác là một hình chữ nhật trên mặt phẳng z=0.

Cuối cùng, **glFlush()** đảm bảo rằng các lệnh vẽ thực sự được thực hiện thay vì lưu trong bộ đệm để chờ thêm các lệnh OpenGL khác. Thủ tục **UpdateTheWindowAndCheckForEvents()** là nơi để đặt các thủ tục quản lý nội dung cửa sổ và bắt đầu xử lý sự kiện.

10.2. Cú pháp lệnh OpenGL

Hầu hết các lệnh trong OpenGL sử dụng tiền tố **gl** và một số từ sau đó thể hiện ý nghĩa của lệnh, được viết hoa ở chữ cái đầu tiên (ví dụ, **glClearColor()**). Tương tự như vậy, OpenGL định nghĩa các hằng bắt đầu bằng GL_, theo sau là các từ sử dụng các chữ cái hoa và ngăn cách nhau bằng kí tự gạch dưới (ví dụ, GL_COLOR_BUFFER_BIT). Đồng thời, OpenGL cũng sử dụng các hậu tố để chỉ kiểu tham số của một hàm. Chẳng hạn hàm **glColor3f()** ám chỉ hàm này có **ba** tham số kiểu số thực (***floating-point numbers***). Có 8 kiểu dữ liệu khác nhau, những chữ cái sử dụng tuân theo những loại kiểu cho ISO C (Xem Bảng 10.1). Tổng quan về cú pháp một lệnh OpenGL được mô tả trong Hình 10.2.

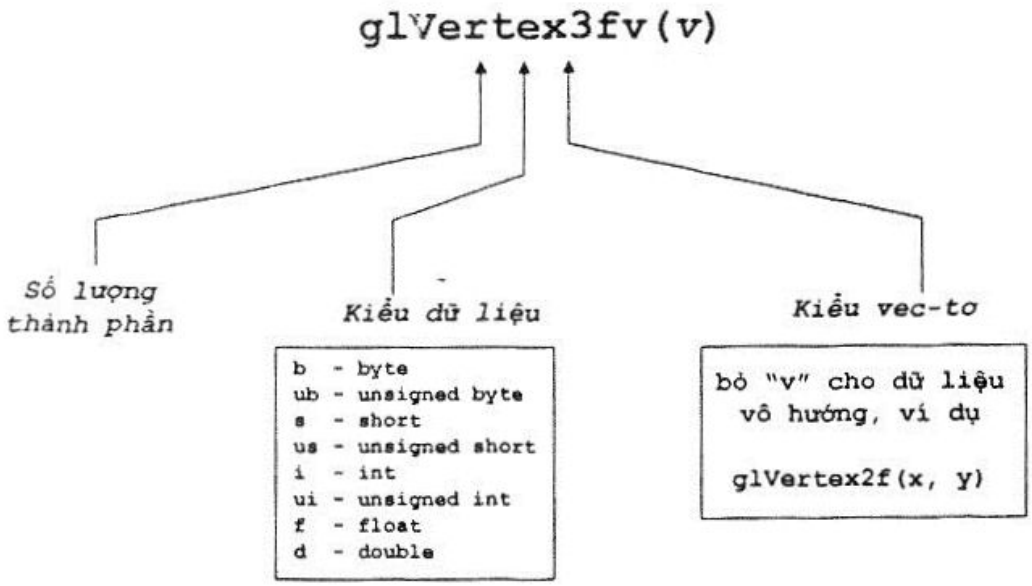
HT	Loại dữ liệu	Kiểu dữ trong C	Định nghĩa trong OpenGL
b	8-bit integer	signed char	Glbyte
s	16-bit integer	short	GLshort
i	32-bit integer	int hoặc long	GLint, GLsizei
f	32-bit floating-point	Float	GLfloat, GLclampf
d	64-bit floating-point	double	GLdouble, GLclampd
ub	8-bit unsigned integer	unsigned char	GLubyte, GLboolean
us	16-bit unsigned integer	unsigned short	GLushort
ui	32-bit unsigned integer	unsigned int hoặc unsigned long	GLuint, GLenum, GLbitfield

Bảng 10.1. Những hậu tố và những loại dữ liệu tham số

Như vậy, hai dòng lệnh

```
glVertex2i(1, 3);
glVertex2f(1.0, 3.0);
```

là tương đương. Dòng thứ nhất xác định tạo độ bằng hai số nguyên 32-bits, và dòng thứ hai xác định tọa độ bằng hai số thực động.



Hình 10.2. Cú pháp một câu lệnh của OpenGL.

10.3. OpenGL như một máy trạng thái

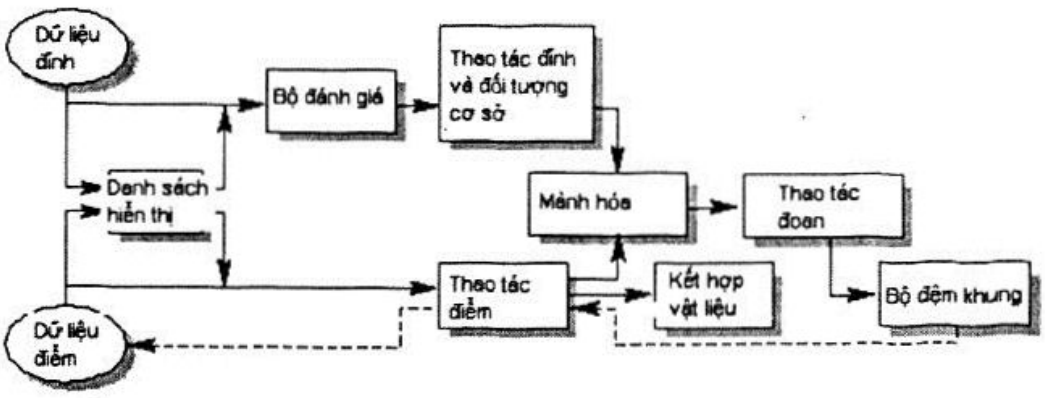
OpenGL thực chất là một máy trạng thái. Chúng ta đặt nó trong rất nhiều trạng thái khác nhau mà chúng được lưu lại cho đến khi chúng ta thay đổi chúng. Chẳng hạn, màu hiện thời là một biến trạng thái. Chúng ta có thể được chúng là màu xanh, đỏ, hoặc những màu khác, và màu hiện thời chỉ thay đổi khi chúng ta đổi chúng. Những biến trạng thái khác của OpenGL gồm có những điều khiển như khung nhìn hiện thời, những phép biến đổi, phép chiếu, kiểu đa giác, các vị trí và tính chất của ánh sáng, và những tính chất vật liệu của đối tượng được vẽ. Rất nhiều biến trạng thái được dùng để điều khiển các trạng thái đó và chúng được kích hoạt và vô hiệu hóa bằng lệnh **glEnable()** hoặc **glDisable()**.

Mỗi biến trạng thái hay chế độ có một giá trị mặc định, và chúng ta có thể truy cập đến giá trị hiện thời của các biến này. Thông thường chúng ta có thể dùng sáu lệnh sau để làm điều đó:

glGetBooleanv(), **glGetDoublev()**, **glGetFloatv()**, **glGetIntegerv()**, **glGetPointerv()**, hoặc **glIsEnabled()**. Chọn lệnh nào được dùng phụ thuộc vào kiểu dữ liệu mà chúng ta muốn nó trả về. Một số biến trạng thái có lệnh truy cập đặc thù hơn (ví dụ như **glGetLight*()**, **glGetError()**, hoặc **glGetPolygonStipple()**). Thêm vào đó, chúng ta có thể lưu trữ một tập các biến trạng thái vào ngăn xếp thuộc tính bằng lệnh **glPushAttrib()** hoặc **glPushClientAttrib()**, tạm thời thay đổi chúng, và sau đó khôi phục lại giá trị bằng lệnh **glPopAttrib()** hoặc **glPopClientAttrib()**. Với những thay đổi trạng thái tạm thời, chúng ta nên sử dụng các lệnh này thay cho các lệnh truy cập khác vì chúng hiệu quả hơn rất nhiều.

10.4. Luồng kết xuất OpenGL

Trong OpenGL, các công đoạn kết xuất được xử lý theo thứ tự, tạo thành một dãy được gọi là **luồng kết xuất OpenGL** (*OpenGL rendering pipeline*). Sơ đồ trong Hình 10.3 cho thấy dữ liệu được xử lý như thế nào trong luồng kết xuất OpenGL. Dữ liệu hình học (đỉnh, cạnh, và đa giác) theo đường mũi tên đến các **bộ đánh giá** (*evaluators*) và sau đó đến các **thao tác hoạt động trên đỉnh** (*per-vertex operations*), trong khi dữ liệu điểm ảnh (điểm và ảnh) lại được xử lý trong phần khác của quá trình xử lý. Cả hai loại dữ liệu đó đều đến những bước kết xuất cuối cùng (mành hóa và thao tác trên đoạn) trước khi dữ liệu điểm ảnh cuối cùng được cập nhật vào trong **bộ đệm khung** (*framebuffer*).



Hình 10.3. Luồng kết xuất OpenGL.

10.5. Những thư viện liên quan đến OpenGL

OpenGL cung cấp một tập lệnh thể hiện đối tượng cơ bản nhưng hiệu quả. Những hình vẽ ở mức trừu tượng cao đều cần phải sử dụng các lệnh đó. Đồng thời, những chương trình OpenGL có thể phải dùng các công cụ để làm việc với hệ thống cửa sổ. Một số thư viện được phát triển cho phép chúng ta đơn giản hóa công việc lập trình để làm những việc trên, bao gồm:

- **Thư viện tiện ích** (*OpenGL Utility Library - GLU*) bao gồm cách sử dụng những câu lệnh OpenGL ở mức độ thấp để thực hiện những công việc ở mức cao hơn, chẳng hạn như khởi tạo những ma trận cho hướng và hình chiếu của tầm nhìn xác định, thực hiện việc tô màu cho đa giác, và phủ bề mặt đối tượng.

- **Bộ công cụ tiện ích** (*OpenGL Utility Toolkit - GLUT*) là một bộ công cụ hệ thống cửa sổ độc lập, được viết bởi Mark Kilgard, để che giấu đi những sự phức tạp của những API hệ thống cửa sổ khác nhau.

10.5.1. Nạp những tệp tin thư viện

Các ứng dụng OpenGL đều phải nạp tệp tin gl.h ở đầu. Ngoài ra, hầu hết các ứng dụng OpenGL đều sử dụng GLU nên đầu các chương trình thường bao gồm những chỉ dẫn nạp tệp thư viện như sau:

```
#include <GL/gl.h>
#include <GL/glu.h>
```

Nếu chúng ta muốn truy cập trực tiếp các thư viện giao diện của hệ điều hành để hỗ trợ OpenGL, chẳng hạn GLX, AGL, PGL, hoặc WGL, chỉ dẫn thêm những dòng sau. Chẳng hạn, nếu gọi GLX, ta cần thêm dòng mã sau:

```
#include <X11/Xlib.h>
#include <GL/glx.h>
```

Nếu chúng ta sử dụng GLUT để quản lý các công việc liên quan đến hệ thống cửa sổ, cần thêm dòng chỉ dẫn sau:

```
#include <GL/glut.h>
```

Lưu ý rằng glut.h mặc nhiên bao gồm gl.h, glu.h, và glx.h, nên khi đã có chỉ dẫn nạp glut.h thì không cần chỉ dẫn nạp 3 tệp kia nữa. Ngoài ra, GLUT đối với hệ điều hành Microsoft Windows còn bao gồm các chỉ dẫn để truy cập WGL.

10.5.2. Bộ công cụ tiện ích OpenGL - GLUT

Như trình bày ở trên, OpenGL được thiết kế bao gồm các lệnh thể hiện đối tượng nhưng lại được thiết kế độc lập với hệ thống cửa sổ và hệ điều hành. Hệ quả là nó không bao gồm những lệnh như mở một cửa sổ hoặc đọc một sự kiện từ bàn phím hay chuột. Chúng ta không thể viết một chương trình nếu không ít nhất một lần mở một cửa sổ và hầu hết các chương trình đều có sự tương tác với dữ liệu vào từ người dùng hoặc từ các dịch vụ khác của hệ điều hành hoặc hệ thống cửa sổ. Thêm vào đó, những lệnh vẽ OpenGL được giới hạn bởi các hình nguyên thủy như điểm, đường hay đa giác. Ngoài các lệnh làm việc với cửa sổ, chuột và bàn phím, GLUT có nhiều cách tạo ra một đối tượng ba chiều phức tạp chẳng hạn như hình cầu, hình xuyến hoặc một ấm trà.

10.5.2.1. Quản lý cửa sổ

Có 5 hàm thực hiện công việc liên quan đến khởi tạo một cửa sổ:

- **glutInit**(int **argc*, char ***argv*) khởi tạo GLUT và xử lý bất kỳ tham số dòng lệnh nào (đối với X, nó có thể là các tùy chọn -display và -geometry). **glutInit()** nên được gọi trước các lệnh GLUT khác.
- **glutInitDisplayMode**(unsigned int *mode*) xác định có hay không việc sử dụng mô hình màu *RGBA* hay bảng màu, cửa sổ có **bộ đệm đơn** (*single-buffered window*) hay **bộ đệm kép** (*double-buffered window*), và cửa sổ có **bộ đệm độ sâu** (*depth buffer*), **bộ đệm stencil** (*stencil buffer*), hay **bộ đệm tích lũy** (*accumulation buffer*). Ví dụ, nếu chúng ta muốn có một cửa sổ với bộ đệm kép, mô hình màu RGBA, và một bộ đệm độ sâu, chúng ta có thể gọi **glutInitDisplayMode**(*GLUT_DOUBLE* | *GLUT_RGB* | *GLUT_DEPTH*).

- **glutInitWindowPosition**(int *x*, int *y*) xác định vị trí của góc trái trên của cửa sổ.
- **glutInitWindowSize**(int *width*, int *size*) xác định kích thước theo điểm ảnh của cửa sổ.
- int **glutCreateWindow**(char **string*) tạo ra một cửa sổ với một ngữ cảnh OpenGL. Nó trả về một định danh duy nhất cho cửa sổ mới.

10.5.2.2. Thủ tục gọi lại hiển thị

glutDisplayFunc(void (**func*)(void)) là một thủ tục đăng ký sự kiện quan trọng nhất và cần gọi đầu tiên. Nó gọi lại thủ tục thể hiện những cái mà chúng ta sẽ nhìn thấy. Bất kỳ khi nào GLUT xác định nội dung của cửa sổ để hiển thị lại, hàm được khai báo bằng **glutDisplayFunc()** được thực thi. Do vậy, chúng ta cần đẩy tất cả những gì cần vẽ vào trong hàm display callback.

Nếu chương trình của chúng ta thay đổi nội dung của cửa sổ, thỉnh thoảng ta sẽ phải gọi lại hàm **glutPostRedisplay**(void), và thường được đặt trong **glutMainLoop()** để hiển thị lại màn hình theo nội dung đã đăng kí trong lần kế tiếp.

10.5.2.3. Chạy chương trình

Thủ tục đăng ký sự kiện cuối cùng mà chúng ta cần gọi là **glutMainLoop**(void). Đây là vòng lặp xử lý các sự kiện của cửa sổ.

Ví dụ 10.2 cho thấy chúng ta có thể sử dụng GLUT để tạo một chương trình được hiển thị trong **Ví dụ 10.1**. Sử dụng GLUT làm tối đa hiệu quả. Những hoạt động khởi tạo chỉ cần gọi một lần được đặt trong thủ tục **init()**. Những hoạt động hiển thị khung cảnh được đặt trong hàm **display()** – là lời gọi lại hàm hiển thị đã được đăng kí qua GLUT.

Ví dụ 10.2

```
#include <GL/gl.h>
#include <GL/glut.h>

void display(void)
```

```
{
/*  Xóa tất cả các điểm  */
    glClear (GL_COLOR_BUFFER_BIT);

/*   Vẽ đa giác màu trắng (hình chữ nhật)với các
đỉnh tại
 *  (0.25, 0.25, 0.0) and (0.75, 0.75, 0.0)
 */
    glColor3f (1.0, 1.0, 1.0);
    glBegin(GL_POLYGON);
        glVertex3f (0.25, 0.25, 0.0);
        glVertex3f (0.75, 0.25, 0.0);
        glVertex3f (0.75, 0.75, 0.0);
        glVertex3f (0.25, 0.75, 0.0);
    glEnd();

/*   thực sự thực hiện các lệnh OpenGL trong bộ đệm
*/
    glFlush ();
}

void init (void)
{
/*  Chọn màu xóa (màu nền)        */
    glClearColor (0.0, 0.0, 0.0, 0.0);

/*  Khởi tạo các giá trị khung nhìn  */
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glOrtho(0.0, 1.0, 0.0, 1.0, -1.0, 1.0);
}

/*
 *  Khai báo kích thước cửa sổ, vị trí, và chế độ
hiển thị
 *  (bộ đệm đơn và chế độ màu RGBA).  Mở cửa sổ với
tiêu đề
 *  "Xin chao". Gọi các thủ tục khởi tạo. Đăng ký
hàm gọi
 *  lại hiển thị. Vào vòng lặp chính và xử lý sự
kiện
 */
int main(int argc, char** argv)
{
    glutInit(&argc, argv);
```

```
    glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB);
    glutInitWindowSize (250, 250);
    glutInitWindowPosition (100, 100);
    glutCreateWindow ("hello");
    init ();
    glutDisplayFunc(display);
    glutMainLoop();
    return 0;
}
```

10.5.2.4. Bắt các sự kiện dữ liệu đầu vào

OpenGL cho phép bắt các sự kiện đầu vào thông qua các thủ tục đăng ký một số hàm gọi lại. Các hàm gọi lại sau khi được đăng ký sẽ được thực thi khi các sự kiện đầu vào xảy ra.

- **glutReshapeFunc**(void (**func*)(int *w*, int *h*)) cho phép đăng ký thủ tục chứa những hành động nào cần được thực hiện khi cửa sổ thay đổi kích thước.
- **glutKeyboardFunc**(void (**func*)(unsigned char *key*, int *x*, int *y*)) và **glutMouseFunc**(void (**func*)(int *button*, int *state*, int *x*, int *y*)) đăng ký thủ bắt các sự kiện từ bàn phím và chuột
- **glutMotionFunc**(void (**func*)(int *x*, int *y*)) đăng ký thủ tục bắt sự kiện bấm chuột trong khi di chuyển chuột.

10.5.2.5. Vẽ đối tượng ba chiều

GLUT có một số thủ tục để vẽ một số đối tượng ba chiều, cụ thể là: **hình nón** (*cone*), **khối hai mươi mặt** (*icosahedron*), **ấm trà** (*teapot*), **khối lập phương** (*cube*), **khối tám mặt** (*octahedron*), **tứ diện** (*tetrahedron*), **khối mười hai mặt** (*dodecahedron*), **hình cầu** (*sphere*), **hình xuyến** (*torus*).

Chúng có thể vẽ những đối tượng dưới dạng khung lưới hoặc là dạng đặc được tô bóng. Ví dụ, những lệnh sau dùng để vẽ hình cầu hoặc hình lập phương:

```
void glutWireCube(GLdouble size);
void glutSolidCube(GLdouble size);
void glutWireSphere(GLdouble radius, GLint
slices, GLint stacks);
```

```
void glutSolidSphere(GLdouble radius, GLint
slices, GLint stacks);
```

10.6. Vẽ các đối tượng hình học

Các hoạt động vẽ hình cơ bản trong OpenGL được chia làm ba loại: xóa cửa sổ, vẽ đối tượng hình học, và vẽ những đối tượng ảnh. Đối tượng ảnh bao gồm các đối tượng như ảnh hai chiều, ảnh bitmap và phông kí tự. Trong phần này chúng ta sẽ đề cập đến cách xóa cửa sổ và vẽ các đối tượng hình học bao gồm điểm, đường thẳng và đa giác phẳng. Những hình phức tạp hơn như các đường cong và mặt cong có thể được vẽ xấp xỉ bằng các đa giác phẳng và đường thẳng.

10.6.1. Xóa cửa sổ

Vẽ trên màn hình máy tính khác nhiều so với vẽ trên tờ giấy trắng. Trên máy tính, bộ nhớ chứa hình ảnh luôn được cập nhật với một hình ảnh cuối cùng được vẽ. Chính vì vậy, nên chúng ta cần xóa sạch nền trước khi vẽ hình mới. Màu mà chúng ta sử dụng cho mầu nền phụ thuộc vào ứng dụng. Chẳng hạn, cho chương trình xử lý văn bản chúng ta dùng màu trắng, còn vẽ hình ảnh nhìn từ phi thuyền không gian, chúng ta có thể xóa nền bằng màu đen trước khi vẽ các ngôi sao, hành tinh hay các phi thuyền khác.

Chúng ta có thể xóa cửa sổ bằng hai phương thức sau:

```
glClearColor(0.0, 0.0, 0.0, 0.0);
glClear(GL_COLOR_BUFFER_BIT);
```

Phương thức thứ nhất đặt màu xóa cửa sổ là màu đen với tham số truyền vào theo kiểu màu RGBA. Phương thức thứ hai xóa cửa sổ với mầu xóa hiện tại. Tham số cho hàm **glClear()** chỉ ra những bộ đệm nào được xóa. OpenGl giữ lại màu xóa hiện thời như biến trạng thái hơn là yêu cầu người dùng xác định lại mỗi khi bộ đệm được xóa. Chẳng hạn, để xóa cả bộ đệm màu và bộ đệm độ sâu, dãy lệnh sau được sử dụng:

```
glClearColor(0.0, 0.0, 0.0, 0.0);
glClearDepth(1.0);
glClear(GL_COLOR_BUFFER_BIT
GL_DEPTH_BUFFER_BIT);
```

Trong trường hợp này, tham số cho lệnh **glClear()** gồm phép OR cho tất cả bộ đệm cần được xóa.

Chi tiết của các hàm đó như sau:

- void **glClearColor**(GLclampf *red*, GLclampf *green*, GLclampf *blue*, GLclampf *alpha*): Đặt màu xóa hiện thời để sử dụng trong xóa bộ đệm theo kiểu RGBA mode. Giá trị alpha nằm trong [0,1]. Màu xóa mặc định là màu đen (0, 0, 0, 0).
- void **glClear**(GLbitfield *mask*): xóa những bộ đệm với giá trị hiện thời. Những bộ đệm được liệt kê trong Bảng 10.2 được kết nối bởi phép OR:

Bộ đệm	Tên
Bộ đệm màu	GL_COLOR_BUFFER_BIT
Bộ đệm độ sâu	GL_DEPTH_BUFFER_BIT
Bộ đệm tích lũy	GL_ACCUM_BUFFER_BIT
Bộ đệm Stecil	GL_STENCIL_BUFFER_BIT

Bảng 10.2. Những bộ đệm được xóa

OpenGL cho phép xóa nhiều bộ đệm cùng một lúc bởi xóa là việc làm tốn nhiều thời gian. Điều này sẽ làm tăng tốc đồ họa với một số phần cứng đồ họa cho phép gán giá trị cho nhiều bộ đệm cùng một lúc. Những phần cứng không hỗ trợ điều này sẽ thực hiện một cách tuần tự.

10.6.2. Vẽ các đối tượng hình học cơ sở

Trong các hệ thống đồ họa thông thường, chúng ta thường làm việc với điểm ảnh trên màn hình như một đối tượng vẽ cơ sở. Tuy nhiên, với OpenGL, vẽ trên màn hình máy tính có sự khác biệt cơ bản. Chúng ta không làm việc với tọa độ vật lý trên màn hình và các điểm ảnh. Thay vào đó, chúng ta làm việc với các tọa độ trong

khối nhìn. OpenGL sẽ lo cho chúng ta những công việc để chiếu cảnh vật 3D mà chúng ta dựng thành ảnh 2D trên màn hình.

Để xác định một điểm 3D trong OpenGL chúng ta sử dụng lệnh *glVertex*. Đây là lệnh cấp thấp nhất trong tất cả các đối tượng hình học nguyên thủy của OpenGL. Lệnh *glVertex* có thể truyền vào từ 1 đến 4 tham số với bất kỳ các kiểu từ **byte** đến **double** theo cách đặt tên hàm. Dòng mã sau xác định một điểm với tọa độ 50 theo trục x, 50 theo trục y và 0 theo trục z.

```
glVertex3f(50.0f, 50.0f, 0.0f);
```

Chúng ta đã có cách xác định một điểm trong không gian vào OpenGL. Vậy chúng ta có thể làm gì với điểm đó, và phải viết lệnh như thế nào trong OpenGL? Một điểm như vậy có thể là điểm cuối của một đường hay một góc của một hình lập phương không? Trong OpenGL, một tập những đỉnh như vậy tạo nên một đối tượng hình học cơ sở, từ một điểm đơn được vẽ trong không gian đến đa giác đóng với số cạnh bất kỳ. Một cách để vẽ những đối tượng hình học cơ sở là sử dụng lệnh `glBegin` để thông báo với OpenGL bắt đầu thể hiện một danh sách các đỉnh của đối tượng đó. Danh sách được kết thúc với lệnh `glEnd`.

Bây giờ chúng ta bắt đầu vẽ với một hình đơn giản nhất là điểm. Hãy xem đoạn mã dưới đây:

```
glBegin(GL_POINTS); // Bắt đầu danh sách các điểm
      glVertex3f(0.0f, 0.0f, 0.0f); // Một điểm
      glVertex3f(50.0f, 50.0f, 50.0f); // Một điểm
khác
glEnd(); // Kết thúc danh sách điểm
```

Tham số cho `glBegin` - `GL_POINTS` thông báo cho OpenGL rằng những đỉnh trong danh sách được hiểu và vẽ như các điểm. Nếu liệt kê nhiều lần bằng nhiều cặp `glBegin` và `glEnd` sẽ lãng phí và thực thi chậm hơn:

```
glBegin(GL_POINTS); // Khai báo một điểm để vẽ
      glVertex3f(0.0f, 0.0f, 0.0f);
glEnd();
glBegin(GL_POINTS); // Khai báo một điểm khác
      glVertex3f(50.0f, 50.0f, 50.0f);
glEnd();
```

Chúng ta cũng có thể đặt kích thước của điểm bằng hàm glPointSize:

```
void glPointSize(GLfloat size);
```

Hàm `glPointSize` có một tham số xác định đường kính xấp xỉ của những điểm ảnh được vẽ ra.

Để vẽ đoạn thẳng, chúng ta sử dụng tham số GL_POINTS cho `glBegin`. Đoạn mã dưới đây vẽ một đoạn thẳng giữa hai điểm (0,0,0) và (50, 50, 50):

```
glBegin(GL_LINES);
      glVertex3f(0.0f, 0.0f, 0.0f);
      glVertex3f(50.0f, 50.0f, 50.0f);
glEnd();
```

Cần lưu ý rằng cứ hai đỉnh xác định một đoạn thẳng. Do đó, với mỗi cặp hai đỉnh được xác định, OpenGL sẽ vẽ một đoạn thẳng. Nếu số đỉnh là lẻ cho GL_LINES thì đỉnh cuối cùng sẽ được bỏ qua. Ví dụ 10.3 thể hiện một ví dụ phức tạp hơn khi vẽ một dãy các đường thẳng mà hai đỉnh của nó nằm dọc và đối nhau trên đường tròn. Kết quả của đoạn mã đó có thể thấy trong Hình 10.4.

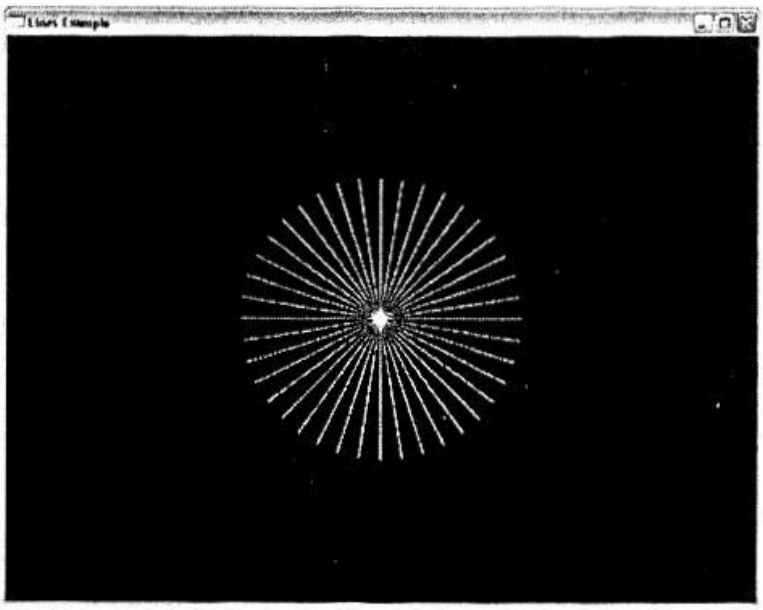
Ví dụ 10.3.

```
// Chỉ gọi một lần cho tất cả các điểm
glBegin(GL_LINES);
// Mọi đoạn thẳng đều nằm trên mặt phẳng xy
z = 0.0f;
for(angle = 0.0f; angle <= GL_PI; angle +=
(GL_PI/20.0f))
{
      // Nửa trên của đường tròn
      x = 50.0f*sin(angle);
      y = 50.0f*cos(angle);
      glVertex3f(x, y, z); // Điểm đầu của đoạn
thẳng
      // Bottom half of the circle
```

```
        x = 50.0f*sin(angle + GL_PI);
        y = 50.0f*cos(angle + GL_PI);
        glVertex3f(x, y, z); // Điểm thứ hai của đoạn
thẳng
}
// Kết thúc danh sách các đoạn thẳng cần vẽ
glEnd();
```

Chúng ta vừa sử dụng GL_LINES cho phép vẽ một danh sách các đỉnh mà thông qua nó từng đoạn thẳng được vẽ ra. Bên cạnh đó, OpenGL dùng GL_LINE_STRIP để vẽ một đường gấp khúc bằng cách xác định các điểm gấp khúc nằm trên nó. Đoạn mã sau đây sẽ vẽ hai đoạn thẳng trên mặt phẳng xy bằng cách xác định 3 đỉnh:

```
glBegin(GL_LINE_STRIP);
        glVertex3f(0.0f, 0.0f, 0.0f); // V0
        glVertex3f(50.0f, 50.0f, 0.0f); // V1
        glVertex3f(50.0f, 100.0f, 0.0f); // V2
glEnd();
```



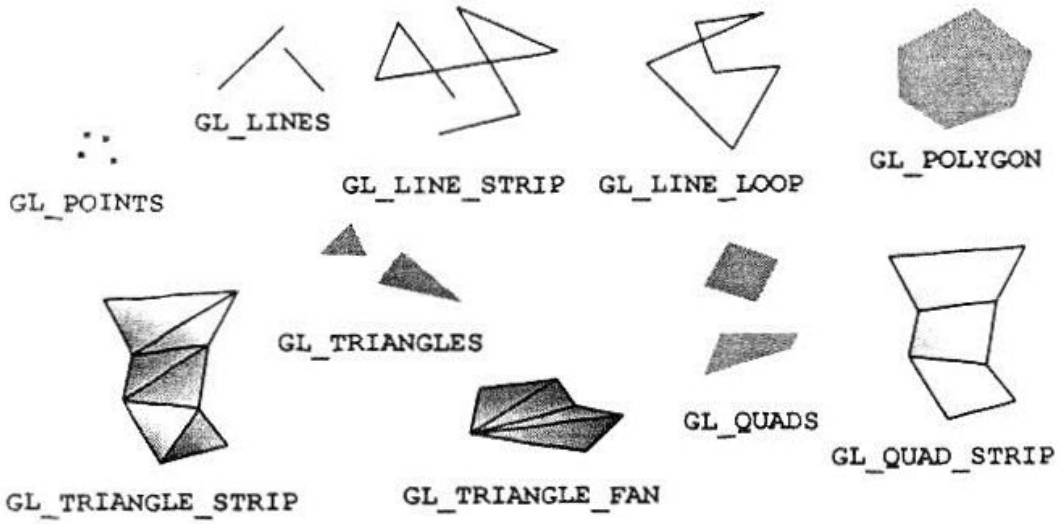
Hình 10.4. Kết quả của chương trình trong Ví dụ 10.3

Cũng như đối với điểm, đối với đoạn thẳng, OpenGL cũng có một hàm đặt độ rộng:

```
void glLineWidth(GLfloat width);
```

Hàm `glLineWidth` có một tham số xác định độ rộng xấp xỉ theo điểm ảnh của đoạn thẳng được vẽ.

Các đối tượng hình học cơ sở khác trong OpenGL có thể thấy trong Hình 10.5.



Hình 10.5. Các đối tượng hình học cơ sở trong OpenGL.

10.7. Quản lý trạng thái OpenGL

Mỗi lần OpenGL xử lý một đỉnh, nó sử dụng dữ liệu được lưu trong các bảng trạng thái bên trong để xác định một đỉnh được chuyển đổi, chiếu sáng, vẽ bề mặt như thế nào. Các thuộc tính được gói trong trạng thái của OpenGL bao gồm:

- Các kiểu kết xuất
- Cách tạo bóng
- Cách chiếu sáng
- Thiết lập ánh xạ

Luồng xử lý kết xuất của OpenGL bao gồm: thiết lập trạng thái cần thiết, truyền đối tượng cơ sở vào để kết xuất, truyền tiếp đối tượng tiếp theo, và cứ tiếp tục như thế. Nói chung, cách thức thông dụng nhất để thao tác với các trạng thái của OpenGL là đặt các thuộc tính cho từng đỉnh, bao gồm: màu sắc, véc-tơ pháp tuyến cho ánh sáng, và tọa độ ảnh chất liệu. Một số lệnh thay đổi thuộc tính của điểm là:

```
glColor*() / glIndex*()
glNormal*()
glTexCoord*()
```

Một số trạng thái OpenGL khác được thiết lập thông qua lệnh glEnable(), với tham số đại diện cho thuộc tính cần thiết lập, ví dụ như GL_LIGHT0 hay GL_POLYGON_STIPPLE:

Thiết lập trạng thái:

```
glPointSize( size )
glLineStipple( repeat, pattern )
glShadeModel( GL_SMOOTH )
```

Kích hoạt thuộc tính

```
glEnable( GL_LIGHTING );
glDisable(GL_TEXTURE_2D);
```

10.8. Các phép biến đổi trong OpenGL

10.8.1. Tổng quan

Ba loại biến đổi (mô hình – khung nhìn, phối cảnh, ảnh chất liệu) là một phần của trạng thái OpenGL. Các thao tác tính toán được thực hiện trong luồng kết xuất (phần khung nhìn) của OpenGL.

Trong OpenGL, một điểm được biến đổi bởi các ma trận 4 x 4, trong đó:

- việc kết hợp giữa các phép biến đổi được thực hiện qua nhân ma trận,
- các ma trận được lưu trữ theo cột,
- các ma trận biến đổi được nhân vào bên phải của ma trận hiện thời, có nghĩa là ma trận biến đổi được mô tả cuối cùng sẽ được áp dụng đầu tiên.

Người lập trình cũng có thể tự thay đổi giá trị cho ma trận biến đổi thông qua lệnh glLoadMatrix và glMultMatrix. OpenGL cũng cung cấp các ngăn xếp ma trận với mỗi loại biến đổi (mô mình – khung nhìn, phối cảnh, ảnh chất liệu) cho phép người lập trình dễ

dáng lưu lại trạng thái biến đổi hiện tại. Các lệnh làm việc với ngăn xếp ma trận là glLoadIdentity(), glPushMatrix() và glPopMatrix(). Để thay đổi ngăn xếp ma trận hiện tại, chúng ta sử dụng lệnh glMatrixMode(GL_MODELVIEW hoặc GL_PROJECTION).

10.8.2. Thay đổi tầm nhìn

Tầm nhìn (viewing) trong OpenGL liên quan đến khung nhìn 2 chiều để xác định vùng hiển thị trên màn hình, khối nhìn 3 chiều để xác định phép chiếu và điểm nhìn. Lưu ý rằng hệ thống thị giác của con người và ống kính máy ảnh có khối nhìn hình nón. OpenGL, cũng như các thư viện đồ họa khác, sử dụng khối nhìn hình kim tự tháp (xem Hình 10.6). Do đó, máy tính sẽ "nhìn" khác với con người, đặc biệt là dọc theo các cạnh của khối nhìn. Với một chương trình OpenGL, chúng ta sẽ phải xác định khung nhìn 2 chiều và khối nhìn 3 chiều. Khung nhìn 2 chiều được xác định với lệnh

```
glViewport( x, y, width, height )
```

và thường với kích thước trùng với kích thước cửa sổ. Ngoài ra, tỉ lệ hai cạnh của khung nhìn cũng nên trùng với tỉ lệ trong phép chiếu, nếu không ảnh kết quả sẽ bị méo.

Đối với khối nhìn 3 chiều, OpenGL cho phép xác định theo hai phép chiếu: chiếu phối cảnh và chiếu trực giao song song. Với phép chiếu phối cảnh, khối nhìn, cũng như phép chiếu đó, được xác định với lệnh:

```
gluPerspective( fovy, aspect, zNear, zFar )
```

hoặc

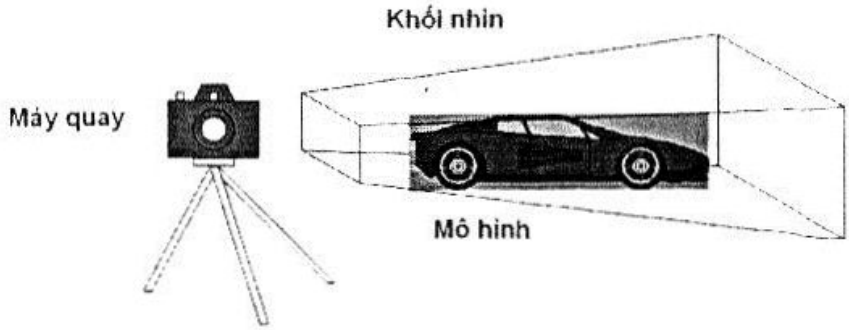
```
glFrustum( left, right, bottom, top, zNear, zFar )
```

Phép chiếu trực giao song song được xác định với lệnh:

```
glOrtho(left, right, bottom, top, zNear, zFar)
```

và

`gluOrtho2D( left, right, bottom, top )` (lệnh này gọi glOrtho với z gần bằng 0)



Hình 10.6. Khối nhìn trong OpenGL với phép chiếu phối cảnh.

Để thiết lập góc nhìn thông qua tham số về máy ảnh/điểm nhìn, OpenGL có lệnh

```
gluLookAt(eyex,eyey,eyez,aimx,aimy,aimz,upx,upy,upz)
```

trong đó `eyex, eyey, eyez` để xác định vị trí của điểm nhìn, `aimx, aimy, aimz` để xác định hướng nhìn và `upx, upy, upz` để xác định hướng lên trên của hướng nhìn. Với lệnh này, OpenGL cho phép người lập trình dễ dàng tạo nên nhiều góc nhìn khác nhau của một cảnh vật chỉ bằng một lệnh mà không cần vẽ lại hay tính toán lại toàn bộ.

10.8.3. Biến đổi mô hình

Với OpenGL, chúng ta có thể dễ dàng có được ma trận **biến đổi mô hình** (*modeling transformation*) bằng cách kết hợp các phép biến đổi cơ bản với nhau như glRotate(), glTranslate(), và glScale(). Cụ thể, để tịnh tiến, chúng ta sử dụng lệnh:

```
glTranslate{fd}( x, y, z )
```

trong đó x, y, z xác định vec-tơ tịnh tiến.

Để quay một vật thể quanh một trục bất kỳ, chúng ta sử dụng lệnh

```
glRotate{fd}( angle, x, y, z )
```

trong đó angle là góc theo độ, và x, y, z xác định trục quay.

Để thực hiện phép co giãn đối với một vật thể, chúng ta sử dụng lệnh:

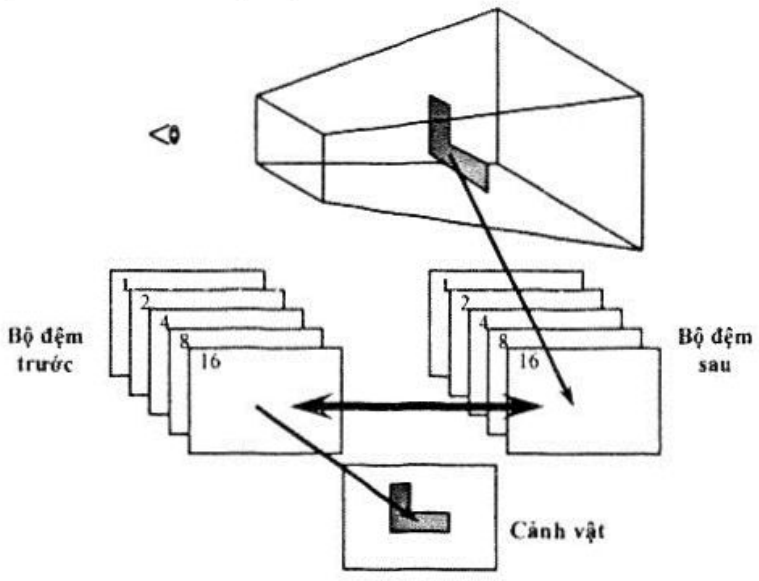
```
glScale{fd}( x, y, z )
```

trong đó x, y, z xác định các hệ số co giãn theo 3 trục.

Bởi vì các ma trận biến đổi là một phần của trạng thái OpenGL, chúng phải được xác định trước khi chúng được áp dụng cho bất cứ điểm nào. Ngoài ra, người ta thường định nghĩa các đối tượng theo hệ tọa độ của chúng và sau đó sử dụng các phép biến đổi của OpenGL để đặt chúng vào cảnh vật.

10.9. Hoạt hình trong OpenGL

Để tạo **hoạt hình** (*animation*), bộ tiện ích GLUT của OpenGL cho phép chúng ta sử dụng chế độ bộ đệm kép để có thể tạo ra các chuyển động mịn, không bị giật. Với chế độ bộ đệm kép, bộ đệm màu được chia làm hai nửa, được gọi là **bộ đệm trước** (*front buffer*) và **bộ đệm sau** (*back buffer*) (xem Hình 10.7). Bộ đệm trước được hiển thị ra màn hình trong khi ứng dụng đang cập nhật bộ đệm sau thông qua các lệnh kết xuất của OpenGL. Khi ứng dụng cập nhật xong bộ đệm sau, nó sẽ yêu cầu phần cứng đổi vai trò của hai bộ đệm, làm cho bộ đệm sau được hiển thị ra màn hình, còn bộ đệm trước lại trở thành bộ đệm sau.



Hình 10.7. Mô hình bộ đệm kép trong OpenGL.

Để yêu cầu OpenGL chuyển sang chế độ bộ đệm kép, chúng ta dùng lệnh

```
glutInitDisplayMode(GLUT_DOUBLE | các chế độ khác);
```

sau đó xóa bộ đệm màu

```
glClear(GL_COLOR_BUFFER_BIT);
```

vẽ cảnh vật, và yêu cầu OpenGL đổi chỗ hai bộ đệm

```
glutSwapBuffers();
```

10.10. Bộ đệm độ sâu

OpenGL thực hiện việc xác định bề mặt hiện (xem Chương 7) thông qua **bộ đệm độ sâu** (*depth buffer*). Để sử dụng chế độ bộ đệm độ sâu, chúng ta đặt chế độ bằng lệnh

```
glutInitDisplayMode(GLUT_DEPTH | các chế độ khác);
```

và sau đó kích hoạt bộ đệm độ sâu

```
glEnable( GL_DEPTH_TEST );
```

xóa bộ đệm màu và bộ đệm độ sâu

```
glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
```

và vẽ cảnh vật.

10.11. Một chương trình hoạt hình đơn giản với OpenGL

Ví dụ 10.4 là một chương trình đơn giản để tạo ra hoạt hình với OpenGL. Trong chương trình này, một tứ giác có màu sắc được vẽ và liên tục được quay quanh trục z theo thời gian. Ví dụ này minh họa cho chúng ta một chương trình OpenGL hoàn chỉnh được tổng hợp của những khái niệm ở các phần trên.

Ví dụ 10.4

```
#include <GL/glut.h>

#define window_width 640
#define window_height 480

// goc quay theo trục z
static float angle;
```

```
// Các lệnh vẽ
void display()
{
      // Xóa bộ đệm màu và bộ đệm độ sâu
      glClear(GL_COLOR_BUFFER_BIT |
GL_DEPTH_BUFFER_BIT);
      // Nạp ma trận biến đổi đơn vị
      glLoadIdentity();
      // Nhân với ma trận tịnh tiến
      glTranslatef(0,0, -10);
      // Nhân với ma trận quay
      glRotatef(angle, 0, 0, 1);
      // Vẽ một tứ giác có màu sắc
      glBegin(GL_QUADS);
            glColor3ub(255, 000, 000);
            glVertex2f(-1, 1);
            glColor3ub(000, 255, 000);
            glVertex2f( 1, 1);
            glColor3ub(000, 000, 255);
            glVertex2f( 1, -1);
            glColor3ub(255, 255, 000);
            glVertex2f(-1, -1);
      glEnd();
      // Đổi chỗ hai bộ đệm
      glutSwapBuffers();
}

// thay đổi thông số hoạt hình
void animate()
```

```
{
      // Tăng góc quay
      angle+=0.25;
      // Yêu cầu vẽ lại
      glutPostRedisplay();
}

// Khởi tạo tầm nhìn tùy theo kích thước cửa sổ
void resizeFunction(int width, int height)
{
      glViewport( 0, 0, width, height );
      glMatrixMode( GL_PROJECTION );
      glEnable( GL_DEPTH_TEST );
      gluPerspective( 45, (float)width/height, 1,
100 );
      glMatrixMode( GL_MODELVIEW );
}

// Chương trình chính
int main(int argc, char** argv) {
      glutInit(&argc, argv);
      glutInitWindowSize(window_width,
window_height);
      glutInitDisplayMode(GLUT_RGB | GLUT_DOUBLE)
      glutCreateWindow("Vi du ve hoat anh");
      glutDisplayFunc(display);
      glutIdleFunc(animate);
      glutReshapeFunc(resizeFunction);
      glutMainLoop();
}
```

Tài liệu tham khảo

1. [Agoston05] Max K. Agoston, "Computer Graphics and Geometric Modeling: Implementations and Algorithms", Springer-Verlag, London, 2005.
2. [Appe67] Appel, Arthur, "The Notion of Quantitative Invisibility and the Machine Rendering of Solids," Proc. ACM Nat. Conf., 1967, 387–393.
3. [Bres65] Bresenham, J.E., "Algorithm for Computer Control of a Digital Plotter," IBM Systems Journal, **4**(1), 1965, 25–30.
4. [Bres77] Bresenham, J.E., "A Linear Algorithm for Incremental Digital Display of Circular Arcs," CACM, **20**(2), February 1977, 100–106.
5. [Catm78] Catmull, Edwin, "A Hidden-Surface Algorithm with Anti-aliasing," SIGGRAPH 78, **12**(3), August 1978, 6–11.
6. [Coon67] Coons, S.A., "Surfaces for Computer Aided Design of Space Forms," MIT Project Mac, TR-41, MIT, Cambridge, MA, June 1967.
7. [Chin95] Chin, Normal, "A Walk Through BSP Trees," Graphics Gems, 1995, 121–138.
8. [CyrB78] Cyrus, M., and Beck, J., "Generalized Two- and Three-Dimensional Clipping," Computers and Graphics, **3**(1), 1978, 23–28.
9. [Fish90] Fishkin, Ken, "Filling a Region in a Frame Buffer," Graphics Gems, 1990, 278–284.
10. [Foley93] James D. Foley, A. van Dam, S. K. Feiner, J. F. Hughes, R. L. Phillips, "Introduction to Computer Graphics", Addison Wesley, 1993.

11. [FuAG83] Fuchs, H., Abram, G.D., and Grant, E.D., "Near Real-Time Shaded Display of Rigid Objects," SIGGRAPH 83, **17**(3), July 1983, 65–69.

12. [FuKN80] Fuchs, H., Kedem, Z.M., and Naylor, B.F., "On Visible Surface Generation by a Priori Tree Structures," SIGGRAPH 80, **14**(3), July 1980, 124–133.

13. [Heck90] Heckbert, Paul S., "Digital Line Drawing," Graphics Gems, 1990, 99–100.

14. [Kapl85] Kaplan, Michael R., "Space-Tracing, a Constant Time Ray-Tracer," Course Notes, Volume 11, SIGGRAPH 85, July 1985.

15. [LiaB83] Liang, You-Dong, and Barsky, Brian A., "An Analysis and Algorithm for Polygon Clipping," CACM, **26**(11), Nov., 1983, 868–877, and Corrigendum, CACM, **27**(2), February 1984, 151.

16. [LiaB84] Liang, You-Dong, and Barsky, Brian A., "A New Concept and Method for Line Clipping," ACM TOG, **3**(1), January 1984, 1–22.

17. [McReynolds05] Tom McReynolds and David Blythe, "Advanced Graphics Programming Using OpenGL", Morgan Kaufmann Publishers, 2005.

18. [NeNS72] Newell, M.E., Newell, R.G., and Sancha, T.L., "A New Approach to the Shaded Picture Problem", Proc. ACM National Conference, 1972.

19. [Newman84] Newman William M. & Sproull Robert F., "Principles of interactive computer graphics", McGraw-Hill, 1984.

20. [Paet95] Paeth, Alan W. (editor), "Graphics Gems V", Academic Press, 1995.

21. [Pitt67] Pitteway, M.L.V., "Algorithm for Drawing Ellipses or Hyperbolae with a Digital Plotter," Computer J., **10**(3), November 1967, 282–289.

22. [Robe63] Roberts, L.G., "Machine Perception of Three Dimensional Solids," Lincoln Laboratory, TR 315, MIT, Cambridge, MA, May 1963.

23. [Roge98] Rogers, David F., "Procedural Elements for Computer Graphics", 2nd Edition, McGraw-Hill, 1998.

24. [Sabi90] Sabin, Malcolm, "Sculptured Surface Definitions - A Historical Survey," in Rogers, D. F. and Earnshaw, R. A. (eds), Computer Graphics Techniques: Theory and Practice, Springer, 1990, 285–337.

25. [SBGS69] Schumacker, R.A., Brand, B., Gilliland, M., and Sharp, W., "Study for Applying Computer-Generated Images to Visual Simulation", Technical Report AFHRL-TR-69-14, NTIS AD700375, U.S. Air Force Human Resources Lab., Air Force Systems Command, Brooks AFB, TX, September 1969.

26. [Smit79] Smith, A.R., "Tint Fill," SIGGRAPH 79, **13**(2), August 1979, 276–283.

27. [SutH74] Sutherland, I.E., and Hodgman, G.W., "Reentrant Polygon Clipping," CACM, **17**(1), January 1974, 32–42.

28. [VanN85] Van Aken, Jerry, and Novak, Mark, "Curve-Drawing Algorithms for Raster Displays," ACM TOG, **4**(2), April 1985, 147–169.

29. [Warn69] Warnock, J., "A Hidden-Surface Algorithm for Computer Generated Half-Tone Pictures", Technical Report TR 4–15, NTIS AD-753 671, Computer Science Department, University of Utah, Salt Lake City, UT, June 1969.

30. [Watk70] Watkins, G.S., "A Real Time Visible Surface Algorithm", Ph.D. thesis, Technical Report UTECCSc-70-101, NTIS AD-762 004, Computer Science Department, University of Utah, Salt Lake City, UT, June 1970.

31. [WeiA77] Weiler, K., and Atherton, P., "Hidden Surface Removal Using Polygon Area Sorting," SIGGRAPH 77, **11**(2), 1977, 214–222.

NHÀ XUẤT BẢN ĐẠI HỌC QUỐC GIA HÀ NỘI

16 Hàng Chuối - Hai Bà Trưng - Hà Nội

Điện thoại: (04) 39714896; (04) 39724770. Fax: (04) 39714899

Chịu trách nhiệm xuất bản:

Giám đốc: PHÙNG QUỐC BẢO

Tổng biên tập: PHẠM THỊ TRÂM

Chịu trách nhiệm nội dung:

Hội đồng nghiệm thu Trường ĐHCN - ĐHQGHN

Người nhận xét: PGS. TS. NGUYỄN ĐÌNH HOÁ

TS. NGUYỄN VIỆT HÀ

Biên tập: LAN HƯƠNG

HỮU NGUYÊN

Chế bản: QUANG HƯNG

Trình bày bìa: NGỌC ANH

ĐỒ HOẠ MÁY TÍNH

Mã số: 1K - 05ĐH2009

In 300 cuốn, khổ 16 x 24 cm tại Công ty Cổ phần Nhà in KHCN

Số xuất bản: 500 - 2009/CXB/12 - 79/ĐHQGHN, ngày 10/6/2009

Quyết định xuất bản số: 05 KH-TN/XB

In xong và nộp lưu chiểu quý II năm 2009.